Karlsruhe Institute of Technology

Programmieren in natürlicher Sprache: Alice-Kontrollstrukturen aus natürlicher Sprache

Bachelorarbeit
von

Andrea Claudia Kohlmann

An der Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation (IPD)

Erstgutachter:	Prof. Dr. Walter F. Tichy
Zweitgutachter:	Prof. Ralf H. Reussner
Betreuender Mitarbeiter:	Dipl. Inform.-Wirt Mathias Landhäußer

Bearbeitungszeit: 01. Juni 2013 – 30. September 2013

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Die Regeln zur Sicherung guter wissenschaftlicher Praxis im Karlsruher Institut für Technologie (KIT) habe ich befolgt.

Karlsruhe, 30. September 2013

. . . . . .

(**Andrea Claudia Kohlmann**)

Kurzfassung

Der Ansatz der natürlichsprachlichen Programmierung soll besonders das Interesse von Nichtinformatikern am Programmieren wecken und ihnen den Einstieg erleichtern. Das Projekt *Programmieren in natürlicher Sprache* hat zum Ziel, ein Programm zu entwickeln, mit dessen Hilfe Benutzer durch drehbuchartige Beschreibungen in natürlicher Sprache Videosequenzen in Alice erzeugen können. Im Rahmen der vorliegenden Arbeit wurde hierzu untersucht, wie Beschreibungen von Alice-Kontrollstrukturen in solchen Texten aussehen könnten. Es wurde ein Plugin für den GoldenGATE-Editor entwickelt, der Schleifenausdrücke für das Alice-Konstrukt *Loop* in natürlichsprachlichen Texten erkennt und annotiert. In der Evaluation konnten mit dem Plugin besonders bei präzise ausgedrückten Schleifen gute Ergebnisse erreicht werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Im Laufe der Zeit wurden – und werden auch weiterhin – immer mehr Programmiersprachen entwickelt. Sie sind elementar für alle Programmierer, da sie es ihnen vereinfachen, dem Computer Anweisungen zu geben. Dies geschieht, indem Sachverhalte ganz genau und Schritt für Schritt beschrieben werden. Sie sind also eine große Hilfe für Informatiker, jedoch gibt es sehr viele verschiedene Programmiersprachen.

Für Fachfremde, die üblicherweise nicht programmieren können, sind Programmiersprachen äußerst kompliziert und abschreckend. Müssen sie jedoch doch einmal programmieren, ist der Aufwand, sich eine Programmiersprache anzueignen und zu lernen, seine Gedanken in Code umzuwandeln viel zu hoch, als dass es sich für sie wirklich lohnen würde.

Eine sehr vielversprechende Möglichkeit, diesem Problem entgegenzuwirken, wäre *Programmieren in natürlicher Sprache*. Man könnte also einfach beschreiben, *was* geschehen soll, anstatt ganz genaue Anweisungen zu geben, *wie* etwas geschehen soll. Das ist ein Ansatz, der beispielsweise mit SQL oder von Gulwani[Gul12] verfolgt wird. Dies wäre sicherlich auch ein Anreiz für Laien, die sich bisher nicht an das Programmieren herangewagt haben.

Mit diesem Ansatz beschäftigt sich das Projekt *Programmieren in natürlicher Sprache*, in das sich die vorliegende Bachelorarbeit eingliedert. Im Rahmen dieses Projekts soll ein Programm entwickelt werden, um mit natürlicher Sprache Alice-Videosequenzen zu erzeugen (siehe Abschnitt 1.1). Diese Bachelorarbeit beschäftigt sich dabei mit der automatischen Erkennung von Kontrollstrukturen, wie zum Beispiel Schleifen, und ihrer Annotation in Textfragmenten. Das Endprodukt dieser Arbeit soll ein Plugin für GoldenGATE [SBPT07] werden.

1.1. Das Gesamtprojetkt „Programmieren in natürlicher Sprache“

Das Gesamtprojekt *Programmieren in natürlicher Sprache* arbeitet auf dem Rahmenwerk Alice [CAB+00], ein besonders an Programmieranfänger gerichtetes Programm, mit dem 3D-Videos und sogar einfache Spiele erzeugt werden können, näheres hierzu in Abschnitt 2.1. Ziel des Projekts ist es, ein Programm zu entwickeln, mit dessen Hilfe Benutzer mittels natürlicher Sprache Alice-Welten programmieren können, wobei es sich hierbei darauf beschränkt, dass aus einer Art Drehbuch eine Alice-Filmsequenz enstehen soll, jedoch keine Spiele oder sonstigen Interaktionen.

Um also eine Alice-Welt anzulegen, soll der Benutzer die Möglichkeit bekommen, in natürlicher Sprache zu beschreiben, wie die Welt aussehen und was geschehen soll. Die Aufgabe der Programms ist es nun, aus einem solchen Beschreibungstext ein Skript zu erzeugen, das die gewünschte Welt möglichst genau erstellt.

Das Gesamtprojekt setzt sich aus mehreren Teilen zusammen, von denen hier jedoch nur die genannt werden. Zu Anfang wurde ein Alice-Korpus erstellt, der 14 Beschreibungstexte sowie eine Musterlösung zu einer Film-Sequenz „Bunny“ umfasst [Ham12], sowie bereits vorhandene Modelle aus Alice in eine Alice-API-Ontologie extrahiert [Pet12] und diese um Synonyme erweitert [Wei12], um die Menge der möglichen Ausdrucksweisen für den Nutzer zu erweitern. Im nächsten Schritt soll nun ein Alice-Handlungsskript erzeugt, das heißt natürlichsprachliche Beschreibungen in Anweisungen für Alice umgewandelt werden.

Dieser Teil des Projektes wurde aufgespalten in mehrere Teilaufgaben. So werden zum Beispiel die Erstellung der Anfangsszenerie, die Verknüpfung des Textes mit den entsprechenden Objekten und Methoden, die Erkennung und Zuordnung benötigter Methodenparameter sowie die zeitliche Ordnung der Beschreibung getrennt bearbeitet, während sich die hier vorgeschlagene Bachelorarbeit ausschließlich mit den Kontrollstrukturen in Alice beschäftigt.

1.2. Aufbau der Arbeit

In Kapitel Kapitel 2 werden einige Grundlagen zum Verständnis der vorliegenden Arbeit bereitgestellt, so zum Beispiel Werkzeuge, die verwendet wurden. Anschließend wird in Kapitel Kapitel 3 ein Überblick über einige mit dieser verwandte Arbeiten gegeben. In Kapitel Kapitel 4 wird das vorliegende Problem analysiert sowie getroffene Entwurfsentscheidungen erläutert. Einen Einblick in die Implementierung des GoldenGATE-Plugins verschafft Kapitel Kapitel 5. Kapitel Kapitel 6 umfasst den letzten Schritt der Arbeit – die Evaluation. Zuletzt werden in Kapitel Kapitel 7 die Ergebnisse der Arbeit zusammengefasst und ein Ausblick auf weiterführende Arbeiten gegeben.

2. Grundlagen

Zum besseren Verständnis der vorliegenden Arbeit bedarf es zunächst einiger Grundlagen, vor allem was das Rahmenarchitektur Alice sowie die bei der Implementierung verwendeten Werkzeuge betrifft. Hier ist vor allem der GoldenGate Editor zu erwähnen, für den im Rahmen der vorliegenden Arbeit ein Plugin entwickelt wurde. Als sehr hilfreiche Werkzeuge stellten sich außerdem der von der Natural Language Processing Group der Stanford University entwickelte Stanford Parser sowie WordNet, eine an der Princeton Univerity entwickelte Synonym-Datenbank, heraus.

2.1. Alice

Alice ist ein an der Carnegie Mellon University entwickeltes Rahmenarchitektur, mit dem man – auch ohne Erfahrung im Bereich von 3D-Grafik-Erzeugung – auf einfache Art und Weise 3D-Welten und sogar Spiele erzeugen kann [CAB+00], [Con97].

Abbildung 2.1 zeigt ein Bildschirmfoto von der Benutzeroberfläche von Alice. Die Bedienung des Programms ist sehr übersichtlich und intuitiv. Zuerst muss der Nutzer die Grundszenerie erstellen. Hierzu kann er per Klick auf den Button "Add" aus einer Vielzahl von Figuren und Gegenständen wählen, die dann in die sogennante Alice-Welt eingefügt werden. Die Figuren werden dann links im Fenster aufgelistet. Ein Objekt besteht hierbei aus mehreren Teilobjekten, eine menschliche Figur beispielsweise besteht aus zwei Armen, Beinen, einem Kopf, etc, die jeweils wieder als eigenes Objekt angesehen werden können, wobei jedoch die Figur selbst ein sogenanntes *First Class Object* ist. (Genaueres hierzu in [Pet12].) Ebenso kann es Beziehungen zwischen verschiedenen First Class Objekten geben. Dies ist zum Beispiel der Fall, wenn die eben genannte Figur etwas in der Hand halten würde. Es ensteht eine Baumstruktur, wobei in obigem Fall das Etwas in der Hand der Figur zwar ihr Kind in dieser Baumstruktur wäre, jedoch im Gegensatz zu den Körperteilen kein Teil der Figur.

Jedes Objekt verfügt über diverse Methoden. Einige hiervon sind standardmäsig für alle Objekte vorhanden, so zum Beispiel die Methode *turn*, andere hingegen sind objektspezifisch. Des Weiteren kann ein Benutzer selbst Methoden anlegen, die er aus bereits bestehenden Methoden zusammensetzt. Während zugunsten der Anfängerfreundlichkeit alles einfach per Drag-and-Drop oder über wenige Mausklicks ausgewählt werden kann, wird in einem separaten Fenster, dem Skript-Editor der zugehörige Programmcode angezeigt. Hier lassen sich auch verschiedene Kontrollstrukturen wie zum Beispiel Schleifen

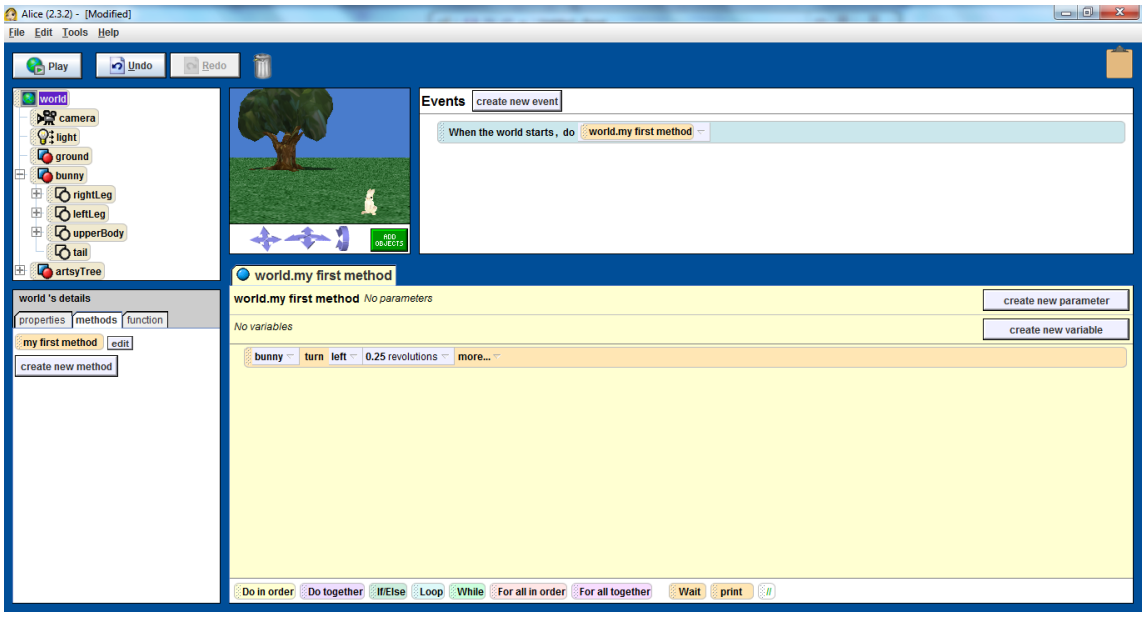

Abbildung 2.1.: Bildschirmfoto der Benutzeroberfläche von Alice

einfügen. Ebenso wurde Alice bewusst so entwickelt, dass komplizierte Notationen oder Formeln vermieden wurden. Beispielsweise müssen keine x-, y- und z-Werte angegeben werden, um Objekte im Raum zu bewegen. Stattdessen kann man intuitiv mit der Maus festlegen, dass ein Objekt nach vorne oder hinten, links oder rechts oder eben oben oder unten bewegt werden soll. Auch gibt es oftmals einfachere und kompliziertere Möglichkeiten, Parameter oder ähnliches festzulegen, sodass der Nutzer sein Können Schritt für Schritt erweitern kann, wie es für ihn persönlich am Besten passt. Mit diesem einfachen Aufbau richtet sich Alice vor allem an Programmieranfänger, insbesondere Kinder, denen das objektorientierte Programmieren auf spielerische Art und Weise näher gebracht werden soll. Ein weiteres Indiz hierfür und zugleich der Hauptgrund, warum im Rahmen des Projekts *Programmieren in natürlicher Sprache* das Rahmenarchitektur Alice verwendet wird, ist der, dass die Namen der Objekte sowie ihrer Eigenschaften und Methoden, etc. in Alice sehr ähnlich wenn nicht sogar gleich derer in der realen Welt sind.

Kontrollstrukturen

Die vorliegende Arbeit beschäftigt sich mit den in Alice angebotenen Kontrollstrukturen. Sie können in Alice am unteren Ende des Skript-Editors ausgewählt werden. Da es nur schwer vorauszusagen war, wieviel Zeit die Bearbeitung der einzelnen Kontrollstrukturen in Anspruch nehmen wird, konnte zu Beginn dieser Arbeit noch keine Aussage darüber getroffen werden, wie viele Kontrollstrukturen sie umfassen würde. Deshalb wurde eine Priotitätenliste angelegt (siehe Abschnitt 4.1) In Tabelle 2.1 wird eine Übersicht über die möglichen Kontrollstrukturen in Alice und ihre Funktionen gegeben werden.

2.2. GoldenGATE

Der am Karlsruher Institut für Technologie entwickelte GoldenGATE Editor ist ein Werkzeug, das die Eigenschaften einfacher XML-Editoren mit denen externer Natural Language Processing Tools vereint. Der Grund hierfür ist, dass NLP-Tools nicht immer hundertprozentig richtig annotieren und man somit oftmals von Hand Verbesserungen vornehmen muss. In GoldenGATE kann man daher sowohl NLP-Tools auf Texte anwenden als auch manuell XML-Annotationen vornehmen, ohne immer zwischen verschiedenen Programmen

Kontrollstruktur	Beschreibung
Do together:	Alle Methoden, die von einem *Do together* umschlossen werden, werden gleichzeitig ausgeführt.
Do in order:	Hierbei werden entsprechend alle umschlossenen Methoden nacheinander ausgeführt. Da dies jedoch die standardmäßige Ausführung von Methoden ist, wird *Do in order* hauptsächlich innerhalb eines *Do togethers* benötigt.
If/else:	Dieses Konstrukt entspricht einer if-else-Abfrage, wie man sie vom Programmieren kennt.
Loop:	*Loop* ist vergleichbar mit einer üblichen for-Schleife. Indem man eine gewünschte Anzahl von Schleifendurchläufen angibt, kann man einen Block mehrfach ausführen lassen.
While:	Wie der Name schon sagt, handelt es sich hierbei um eine gewöhnliche while-Schleife, mit deren Hilfe man einen Block so oft ausführen lassen kann, bis eine bestimmte Bedingung nicht mehr erfüllt ist.
For all together:	Der Methodenblock im Rumpf von *for all together* wird für alle Elemente einer zuvor festgelegten Liste gleichzeitig ausgeführt, wobei die Methoden innerhalb des Blocks nacheinander ausgeführt werden. Die Elemente der Liste sind alle vom selben Typ. Dies können Zahlen, Objekte, Strings, Booleans und viele mehr sein. Die vorliegende Arbeit wird sich jedoch auf Listen von Objekten beschränken, da nur diese im Kontext von Drehbüchern relevant sind. Denkbar wäre zum Beispiel, dass man eine Liste von Hund-Objekten anlegt und sie über dieses Konstrukt alle gleichzeitig bellen lässt.
For all in order:	Bei *for all in order* wiederum werden der Methodenblock innerhalb des Rumpfes für alle Objekte einer Liste nacheinander ausgeführt. Um Missverständnisse vorzubeugen, sei noch zu erwähnen, dass hierbei für ein Listenobjekt nach dem anderen die komplette Methodenabfolge innerhalb *For all in order* ausgeführt wid.

Tabelle 2.1.: Übersicht über die Kontrollstrukturen in Alice

zu wechseln, was erheblichen Aufwand und Zeit erspart. Wie das Bildschirmfoto in Abbildung 2.2 verdeutlicht, kann man beispielsweise eine .txt-Datei öffnen, bearbeiten, manuell mit XML-Tags versehen oder aber – was für diese Arbeit besonders wichtig ist – mit Hilfe verschiedenster Plugins analysieren und annotieren lassen. Ebenso ist es möglich, bereits vorhandene XML-Dateien zu erweitern oder zu verändern.

In Abbildung 2.2 wurde der in Abschnitt 2.4 genauer erläuterte Stanford Parser auf ein .txt-Dokument angewendet, das den Satz „The bunny hops two times.“ enthält. Zugunsten der Übersichtlichkeit kann man durch Setzen beziehungsweise Entfernen der Häkchen in den Kontrollkästchen rechts im Fenster Tags eines bestimmten Annotationstyps anzeigen oder ausblenden lassen. Im betrachteten Fall werden alle Tags des Typs *SyntaxTreeNode* angezeigt, während diejenigen des Typs *Dependency* ausgeblendet sind. Durch einen Klick auf eine bestimmte Markierung kann man außerdem ein neues GoldenGATE-Fenster öffnen, das genau den Teil des annotierten Dokuments anzeigt, der von eben dieser Markierung umschlossen wird. Einen genaueren Überblick über die Funktionen des GoldenGATE-Editors sowie sonstige Informationen kann man sich in [SBPT07] verschaffen.

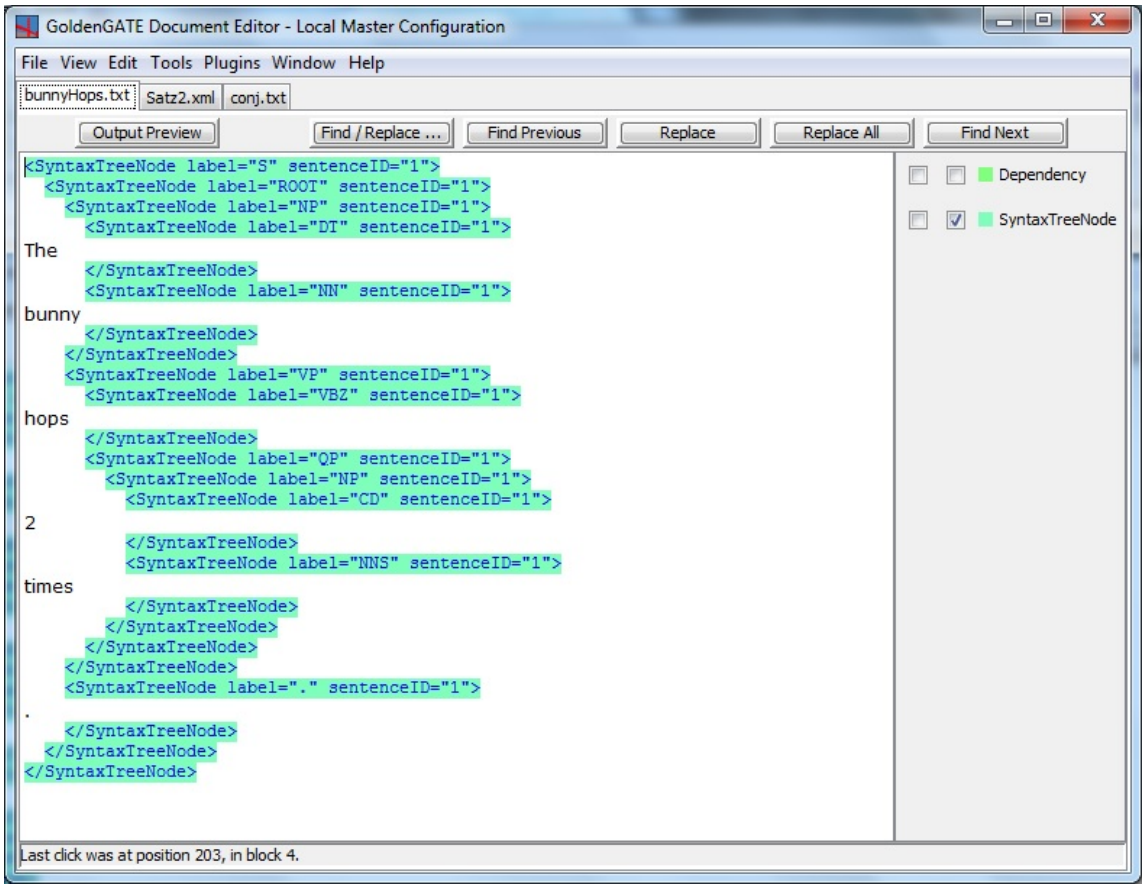

Abbildung 2.2.: Bildschirmfoto der Benutzeroberfläche des GoldenGATE Editors

2.3. Gamta

Gamta ist eine ebenfalls am Karlsruher Institut für Technologie entwickelte Java-Bibliothek, die viele für die Entwicklung eines GoldenGATE-Plugins relevante Funktionen enthält, von denen im Folgenden jedoch nur die wichtigsten der verwendeten näher erläutert werden sollen.

Ein XML-Dokument kann durch ein Objekt der Klasse *MutableAnnotation* repräsentiert werden, die zusätzlich zu den Funktionen aus ihrer Superklasse *QueriableAnnotation* auch noch Funktionen bietet, um diese Nachbildung des Dokuments zu verändern. Das bedeutet diese Veränderungen werden nicht gleich am Dokument selbst vorgenommen, sondern vorerst nur an dieser Nachbildung. Auf diesem Objekt können nun diverse Methoden angewendet werden, um beispielsweise nur bestimmte Teile des Dokuments zu erhalten und gegebenenfalls zu bearbeiten.

Ein weiteres wichtiges Paket von Gamta ist *GPath*. Wie der Name schon andeutet, ist GPath angelehnt an der XML Path Language XPath, jedoch speziell für die Verwendung mit Gamta. Stellt man sich ein Dokument beziehungsweise – im Zusammenhang mit Gamta – ein Objekt der Klasse *QueriableAnnotation* als Baum vor, so bildet ein GPath-Ausdruck einen Pfad über verschiedene verschachtelte Annotationen hin zu einem bestimmten Teil des Dokuments. Damit kann man mit Hilfe eines GPath-Ausdrucks ein *QueriableAnnotation*-Objekt nach allen Dokumentteilen durchsuchen, zu denen ein solcher Pfad führt.

2.4. Stanford Parser

Der Stanford Parser [KM03] ist ein an der Stanford University entwickeltes Modell eines faktorisierten Parsers für natürliche Sprache. Als solcher analysiert er die Satzstruktur natürlichsprachlicher Texte, zerlegt die Sätze in ihre Bestandteile und erstellt dazu einen sogenannten Syntaxbaum, der die Zusammenhänge der einzelnen Bestandteile des Satzes sowie die Wortarten angibt. Hierzu kombiniert er zwei Komponenten, je eine für die Syntax, die sich mit der Struktur eines Satzes beschäftigt, und die Semantik, die sich mit der Bedeutung eines Satzes beschäftigt: Der verwendete PCFG (=probabilistic context-free grammar) Parser für die syntaktische Analyse kombiniert mehrere verschiedene Modelle eben solcher Parsern. Für die semantische Struktur wird eine Menge von Modellen von Parsern für lexikalische Abhängigkeiten verwendet. Durch diese Kombination können genauere Ergebnisse erzielt werden, als bei der Verwendung nur einer dieser Komponenten.

Im Unterschied zu anderen kombinierten Parsern, sind die beiden Komponenten beim Stanford Parser nicht zusammengefasst, sondern liegen getrennt vor, wodurch es einfacher ist, diese zu erweitern.

Wie Klein und Manning beschreiben, wird ein A*-Algorithmus verwendet, um aus den Ergebnissen, die durch die beiden Parser erzielt wurden, das wahrscheinlichste Gesamtergebnis schnell und exakt zu ermitteln.

Die Markierungen der Satzteile und Wortarten, die im Stanford Parser Anwendung finden, basieren auf der Penn Treebank [MMS93]. Die Wortartmarkierungen sind in Unterabschnitt B.1, die Satzteilmarkierungen in Unterabschnitt B.2 angegeben.

Ali Seker hat im Rahmen seiner Arbeit [Sek14] ein GoldenGATE-Plugin für den Stanford Parser geschrieben, das in dieser Arbeit verwendet wird. Dieses enthält zwei Annotationstypen, zum Einen *SyntaxTreeNode* mit den Attributen *label*, in dem die Typen der Satzteile beziehungsweise auf Wortebene die Part-Of-Speech-Tags angegeben werden, und *sentenceID*, eine Identifikationsnummer, mit deren Hilfe genau nachvollzogen werden kann, welche Annotationen zum selben Satz gehören, und zum Anderen *Dependency*, von dem für diese Arbeit vor allem das Attribut *typedDependencies* von Bedeutung ist, in dem die Abhängigkeiten der einzelnen Wörter von einander angegeben werden.

Dieses Plugin wird im Rahmen dieser Arbeit dazu genutzt, bestimmte Schlüsselwörter mit Hilfe ihrer Annotationen einfacher zu finden (siehe Unterabschnitt 4.3.2).

2.5. WordNet

Wordnet ist eine an der Princeton University entwickelte lexikalische Datenbank für englische Wörter, die im Gegensatz zu üblichen Lexika, speziell für die Verwendung in der Programmierung geschaffen wurde. So werden Wörter nicht einfach nur beschrieben, sondern vor allem durch Synonyme erläutert. Um also die Bedeutung eines Wortes anzugeben, werden in WordNet sogenannte „Synsets“ angegeben, wobei ein Wort je nachdem wie viele Bedeutungen es besitzt, in mehreren Synsets vorkommen kann.

Neben Synonymen kennt WordNet auch noch andere semantische Beziehungen zwischen Wörtern. So ist es beispielsweise ebenfalls möglich, Antonyme (Gegenteile), Hyponyme (Unterbegriffe) bzw. Hyperonyme (Oberbegriffe) und Meronyme und Holonyme (Teil-Ganzes-Beziehungen) auszulesen, wobei für die vorliegende Arbeit ausschließlich Synonyme von Bedeutung sind. Eine weitere Eigenschaft WordNets ist die Trennung nach den Wortarten Nomen, Verb, Adjektiv sowie Adverb. Das ist wichtig, da es bekanntlich viele Wörter gibt, die, abhängig vom entsprechenden Kontext, verschiedenen Wortarten angehören können. Ein Beispiel hierfür ist das Wort „kind“, wie ein Bildschirmfoto der grafischen

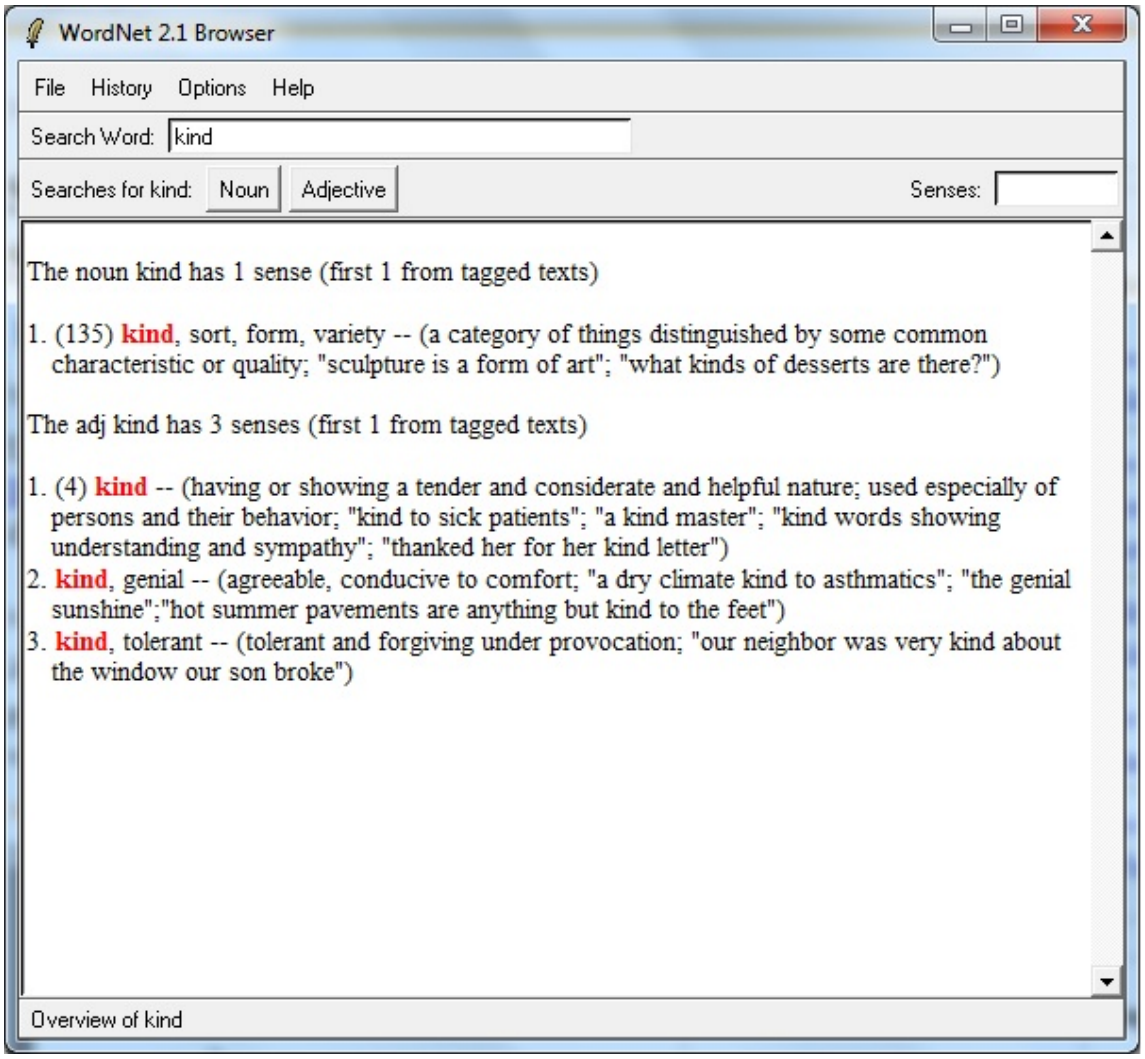

Abbildung 2.3.: Bildschirmfoto der Benutzeroberfläche von WordNet

Nutzeroberfläche (Abbildung 2.3) zeigt. Hier wird ein Synset für das Wort „kind“ als Substantiv und drei weitere für die Verwendung als Verb angegeben. In jedem Synset stehen nun, falls vorhanden, Synonyme zu dem gesuchten Wort sowie die gemeinsame Bedeutung dieser.

Erwähnenswert ist weiterhin die Tatsache, das Wörter nicht zwangsweise in ihrer Grundform verwendet werden müssen. Bei Nomen im Plural oder konjugierten Verben beispielsweise wird neben ihrer eigenen Bedeutung – falls vorhanden – auch die Bedeutung ihrer Grundform angegeben. In dieser Arbeit wird Version 2.1 von WordNet verwendet, um zu ausgeschriebenen Zahlen die entsprechende Zifferndarstellung zu erhalten (siehe Unterunterabschnitt 5.2.1.1).

3. Verwandte Arbeiten

Natürlichsprachliche Programmierung ist ein sehr gefragtes Gebiet, zu dem es bereits zahlreiche Arbeiten gibt, die jedoch trotzdem sehr unterschiedliche Ziele verfolgen und auch sehr unterschiedliche Lösungsansätze verwenden. In diesem Kapitel soll ein Überblick über bereits existierende Arbeiten gegeben werden, die sich mit ähnlichen Ansätzen wie das Projekt *Programmieren in natürlicher Sprache*, in das sich die vorliegende Arbeit eingliedert, befassen.

3.1. Natural Java

Natural Java [PRZH00] ist ein Modell einer Benutzeroberfläche, um Javaprogramme mittels natürlicher Sprache zu erstellen und bearbeiten. Hierbei gibt der Benutzer seine Anweisungen in natürlicher Sprache an und bekommt in einem separaten Fenster den daraus resultierenden Quelltext angezeigt.

Anders als im Projekt *Programmieren in natürlicher Sprache* gibt der Benutzer von *Natural Java* nicht direkt ein, was letztlich geschehen soll, sondern wie es geschehen soll. Er beschreibt in seinen Anweisungen also den Quelltext in vollständigen natürlichsprachlichen Sätzen. Wie Price et al. erläutern, kann eine *for*-Schleife beispielsweise mit dem Satz „Create a for loop that iterates from 1 to 10.“ beschrieben werden. Der Benutzer von *Natural Java* kommt also nicht komplett ohne Java-Kenntnisse aus, wo hingegen es Ziel des Projekts *Programmieren in natürlicher Sprache* ist, dass der Benutzer keinerlei Wissen über Programmierung benötigt, sondern vielmehr eine Art Drehbuch verfassen kann.

3.2. Metafor

Im Rahmen ihrer Arbeit [LL05] haben Liu und Lieberman *Metafor* entwickelt, einen Editor, der eine Geschichte in Python-Quelltext umwandelt. Der Nutzer von Metaphor schreibt hierbei eine Art Geschichte über die Objekte des zu entwickelnden Programmms, zu der gleichzeitig in einem anderen Fenster Quelltext-Fragmente angezeigt werden.

Metafor hat nicht zum Ziel, alle grammatikalisch und sprachlich korrekten Formulierungen zu verstehen und einen direkt ausführbaren Quelltext auszugeben, stattdessen soll er vielmehr als Hilfe beim Entwurf eines Programmes dienen. Dieser Ansatz soll das Programmieren für Anfänger zugänglicher machen sowie fortgeschrittenen Programmierern eine Brainstorming-Plattform bieten.

Metafor ähnelt dem Projekt *Programmieren in natürlicher Sprache* insofern, als beide eine Eingabe in Form einer Geschichte voraussetzen, wobei *Metafor* hierbei weniger Kenntnisse über die Bedeutung eines Wortes benötigt und auch nur ein mögliches Gerüst eines Quelltextes erzeugt. So geht es bei *Metafor* hauptsächlich darum, beispielsweise Nominalphrasen in Objekte, Verben in Methoden und Adjektive in Eigenschaften umzuwandeln. Bei *Programmieren in natürlicher Sprache* hingegen spielt auch die Bedeutung eine große Rolle, da später entsprechende in Alice verfügbare Instanzen ausgewählt werden müssen und in der Alice-Welt agieren sollen, wozu ein vollständiger Quelltext einer Alice-Videosequenz erzeugt werden muss.

3.3. Translating Keyword Commands into Executable Code

Little und Miller beschreiben in ihrer Arbeit [LM06] eine Technik, um vor allem Nichtprogrammierern das Erstellen von Skripten für Applikationen zu erleichtern. Diesen Ansatz haben sie bereits in Form je eines Prototypen für einen Internetbrowser und für Microsoft Word umgesetzt, wobei die Testpersonen einer Studie die ihnen gestellten Aufgaben zu 90% umsetzen konnten, ohne mit dem Erstellen von Skripten vertraut zu sein. Näheres zu dieser Studie findet sich in [LM06].

Bei diesem Ansatz wird dem Nutzer eine Oberfläche geboten, in der er Anweisungen intuitiv durch Schlüsselwörter angeben kann, die anschließend in ausführbaren Quelltext umgewandelt werden. So wird hierbei weder eine korrekte Grammatik, noch irgendeine Art von Zeichensetzung benötigt, jedoch sollte der Nutzer mit dem Vokabular der jeweiligen Applikation vertraut sein. Es werden jedoch wie auch bei *Programmieren in natürlicher Sprache* Synonyme erkannt.

Da jedoch bei Little und Miller nur Schlüsselwörter verwendet werden, können hierbei Mehrdeutigkeiten auftreten. Besitzt eine Funktion beispielsweise mehrere Parameter des gleichen Typs, ist es schwierig, die zugehörigen Schlüsselwörter dem richtigen Parameter zuzuweisen. Um dies zu vermeiden, hat der Nutzer zwar noch weitere Möglichkeiten, seine Anweisungen zu spezifizieren, zum Beispiel durch Setzen von Anführungszeichen, wozu er jedoch wieder über gewisse Konventionen Bescheid wissen muss. Im Projekt *Programmieren in natürlicher Sprache* soll der Nutzer hingegen vollständige Sätze schreiben, durch die derartige Mehrdeutigkeiten ausgeschlossen werden können und auch kaum Konventionen beachtet werden müssen.

4. Analyse und Entwurf

Bevor mit der Implementierung begonnen werden kann, gilt es zuerst, das vorliegende Problem zu analysieren und einen Plan zu entwerfen, wie die Umsetzung in etwa vonstatten gehen soll. Hierzu wurde das Test-Video *Bunny* sowie der zugehörige Text-Korpus aus [Ham12] auf das Vorkommen der verschiedenen Kontrollstrukturen untersucht. Da jedoch nicht alle Kontrollstrukturen im Video und somit im Korpus verwendet werden und es außerdem noch viel mehr mögliche Formulierungen gibt als diejenigen in den Beschreibungstexten, mussten noch weitere Überlegungen angestellt werden, wie Kontrollstrukturen ausgedrückt werden können. Eine Übersicht über mögliche Schlüsselwörter gibt die Tabelle im Anhang A.

In Abschnitt 4.1, wird die besagte Prioritätenliste erstellt und erläutert. Anschließend werden in Abschnitt 4.2 mögliche natürlichsprachliche Formulierungen der Kontrollstrukturen gesammelt und analysiert. Die darauf folgenden Abschnitte nehmen genaueren Bezug auf die umgesetzten Kontrollstrukturen.

4.1. Prioritätenliste

Wie bereits erwähnt, ist es schwierig, schon zu diesem Zeitpunkt festzulegen, wie viele und welche der in Alice verfügbaren Kontrollstrukturen im Rahmen dieser Arbeit umgesetzt werden können. Daher wurde eine Prioritätenliste erstellt, die dann nach und nach abgearbeitet werden soll (siehe Tabelle 4.1).

Zu diesem Zweck wurde der Text-Korpus analysiert und anschließend die Verwendungshäufigkeit sowie die Schwierigkeit bei der Umsetzung der jeweiligen Kontrollstruktur beurteilt beziehungsweise, da nicht alle Kontollstrukturen im Text-Korpus Anwendung fanden, geschätzt.

Hierbei ging hervor, dass die am häufigsten verwendete Kontrollstruktur *Loop* ist, gefolgt von den etwas komplizierteren *Do together* und *Do in oder*. Die beiden letzteren sind vor allem deshalb schwieriger, da sie sich in einem natürlichsprachlichen Text über mehrere Sätze ziehen können und nicht immer eindeutlig feststellbar ist, wo sie aufhören. *Loop* ist außerdem voraussichtlich die am einfachsten umsetzbare Kontrollstruktur. Somit bildet sie den Anfang der Liste.

Do in oder und *do together* werden ungefähr gleich schwierig eingeschätzt. Da jedoch *do in order* hauptsächlich innerhalb von *do together* benötigt wird, ist es sinnvoll, letzteres zuerst zu bearbeiten.

Priorität	Kontrollstruktur	Verwendungshäufigkeit	Schwierigkeit
1	Loop	1	1
2	Do together	2	4
3	Do in oder	3	4
4	For all together	4	5
5	For all in order	4	5
6	While	5	3
7	if/else	6	2

Tabelle 4.1.: Liste der umzusetzenden Kontrollstrukturen mit von oben nach unten sinkender Priorität und Werten, die die Verwendungshäufigkeit und Umsetzungsschwierigkeit angeben. Hierbei gilt: kleine Zahl = hohe Priorität/große Verwendungshäufigkeit/geringe Schwierigkeit.

For all together und *For all in order* werden in den natürlichsprachlichen Texten später vermutlich eher selten verwendet und wurden aufgrund der benötigten Listen als schwieriger befunden. Sie wurden daher nach *do in order* in die Liste eingeordnet.

While wird zwar als weniger schwierig eingeschätzt, wird jedoch trotzdem an die vorletzte Stelle der Prioritätenliste gesetzt, da es wahrscheinlich zusammen mit *if/else* am seltesten benötigt wird, denn der Nutzer müsste hierfür innerhalb seines Beschreibungstextes eine Bedingung festlegen, wie lange etwas geschehen soll. Meist wird er jedoch genau wissen, wie lange es geschehen soll und damit ein entsprechendes Setzen von Parametern ermöglichen, was eine Bedingung hier überflüssig machen wird. Dennoch sollte *while* umgesetzt werden, denn es gibt doch einige Fälle, in denen eine while-Schleife sehr nützlich sein dürfte. So zum Beispiel, wenn ein Objekt bis zum Rand des Bildes bewegt werden soll. In diesem Fall müsste die Position dieses Objekts mit den Koordinaten des Bildrandes verglichen werden, um zu verhindern, dass das Objekt aus dem Bild verschwindet.

If/else soll vorerst nicht weiter betrachten werden und bildet daher das Ende der Prioritätenliste, auch wenn es wahrscheinlich eher einfach umzusetzen wäre. Dies liegt daran, dass diese Kontrollstruktur im Normalfall von Drehbuchautoren nicht benötigt wird. Ein Benutzer gibt in seinem Beschreibungstext schließlich genau an, was passieren soll.

4.2. Analyse natürlichsprachlicher Formulierungen

Zu Beginn der Entwurfsphase wurden für alle Kontrollstrukturen Schlüsselwörter gesammelt, um eine Vorstellung davon zu bekommen, wie ein Benutzer eine Kontrollstruktur in einem Beschreibungstext verwenden würde. Eine Übersicht über mögliche Schlüsselwörter ist in Abschnitt A des Anhangs zu finden. Einige dieser Schlüsselwörter, vor allem diejenigen, die auf ein Nacheinander hinweisen, finden sich auch in der Arbeit von Tobias Hey [Hey13], die sich mit der zeitlichen Sortierung von Beschreibungstexten beschäftigt. Im den folgenden Unterabschnitten wird nun auf die unterschiedlichen Kontrollstrukturen und ihre Schlüsselwörter eingegangen.

4.2.1. if/else

Eine if/else-Bedingung tritt im Text-Korpus nicht auf. Auch allgemein, kann man davon ausgehen, dass diese Kontrollstruktur im Normalfall von Drehbuchautoren nicht benötigt wird, schließlich gibt ein Benutzer in seinem Beschreibungstext genau an, was passieren soll und benötigt somit keine Bedingungen.

4.2.2. While

Auf eine *while*-Schleife kann natürlich das Wort „while“ hinweisen, das jedoch ebenfalls ein Schlüsselwort für *Do together* ist (siehe Unterabschnitt 4.2.6). Zu unterscheiden ist hierbei, ob „while“ angibt, dass eine Aktion während einer anderen ausgeführt wird, was dann auf ein *Do together* hindeutet, oder ob es angibt, dass eine Aktion ausgeführt wird, solange eine bestimmte Bedingung erfüllt ist, also zum Beispiel solange eine bestimmte Variable *true* ist. In letzterem Fall würde eine while-Schleife benötigt. Gleiches gilt für *during* und *as long as*. Ein weiterer, eindeutigerer Begriff wäre *until* oder *till*. Auch *before* kann auf eine *while-Schleife* hindeuten, so zum Beispiel in dem Satz „The bunny hops all the time before the frog begins to ribbit.“. Die durch diesen Satz implizierte *while*-Schleife würde den Hasen also die ganze Zeit hüpfen lassen, bis der Frosch zu quaken anfinge.

In jedem Fall sollte im Text auch eine Bedingung erwähnt werden. Es ist jedoch auch möglich, dass diese gar nicht gleich als solche erkannt wird. Denkbar wäre zum Beispiel der Satz „The bunny hops to the border of the screen.“. In diesem Fall wird weder ein klares Schlüsselwort verwendet, noch eine eindeutig erkennbare Bedingung. Jedoch könnte man damit bewirken, dass die Methode *hop* so lange ausgeführt wird, bis die Koordinaten des Hasen einen bestimmten Randwert erreicht haben. Eine andere Möglichkeit wäre es allerdings auch, dies nicht mittels *while*-Schleife, sondern vielmehr durch bestimmte Parameter sicherzustellen.

4.2.3. For all in order

Die Schlüsselwörter von *for all in order* sind zum einen Wörter wie *all, every, any, a group/herd/pride of, etc*, die auf eine Gruppe von Objekten hinweisen, *one by one, one at a time, one after another*, die ein Nacheinander bezüglich der Objekte implizieren und natürlich Begriffe, die die Abfolge von Methoden innerhalb des Rumpfes angeben wie zum Beispiel *then, next, subsequently*. Letztere entsprechen derer von *do in order*.

Ein Fall einer Liste für *for all in order* wäre eine, die mehrere gleiche Objekte enthält, beispielsweise ein Rudel Hunde, was durch Sätze wie „One by one all dogs walk two meters.“ ausgedrückt werden kann. In diesem Beispiel müssten alle Hunde der Alice-Welt nacheinander loslaufen. Des Weiteren wäre es denkbar, dass mehrere verschiedene Objekte etwas nacheinander tun sollen wie in „The bunny, the frog, the penguin and the two dogs one by one turn to face the sunflower.“

Eine Schwierigkeit besteht nun darin, die Objekte, die in die für *for all in order* benötigte Liste gehören, im Text zu erkennen und die Liste anzulegen. Einfach wäre dies, wenn der Benutzer gleich zu Beginn des Textes angibt, dass mehrere Objekte in eine Liste gehören. Jedoch wird dies wahrscheinlich eher selten der Fall sein, da es sich bei den Benutzern ja hauptsächlich um Programmieranfänger handelt, die keine Vorstellung davon haben, wie ihr Text umgewandelt wird. Es ist eher anzunehmen, dass gar nicht erwähnt wird, dass eine Liste erstellt werden soll, was bedeutet, dass erkannt werden muss, in welchen Fällen man eine Liste erstellen sollte und in welchen man eine Nacheinander-Aktion einfach mittels *do in order* verwirklicht. Eventuell wäre es hierzu sinnvoll, beispielsweise festzulegen, dass eine Liste nur dann angelegt wird, wenn diese Gruppe von Objekten öfter in der selben Konstellation auftritt. Außerdem könnte man eine Anzahl an Objekten festlegen, ab welcher man *for all in order* verwendet. Für nur zwei Objekte wäre es zum Beispiel nicht nötig eine Liste anzulegen. Ebenso macht es erst dann Sinn, dieses Konstrukt zu verwenden, wenn die Gruppe mehr als nur eine Methode ausführen soll.

4.2.4. For all together

For all together hat dementsprechend ebenso die Schlüsselwörter von *for all in order*, die auf eine Gruppe hinweisen, und dann natürlich Begriffe, die die Gleichzeitigkeit angeben

wie zum Beispiel *at the same time, together, simultaneously.* Ein Beispielsatz, aus dem ein *for all together* folgen würde, wäre entsprechend „All dogs bark simultaneously.“.

Die Angaben über die Schwierigkeit, eine benötigte Liste im Text zu erkennen und anzulegen, lassen sich von *for all in order* auf *for all together* übertragen.

4.2.5. Do in order

Schlüsselwörter, die eindeutig auf ein Nacheinander, also auf ein *do in order* hinweisen, können beispielsweise *then, after that, after this, afterwards, later, subsequently, next, at last, finally* sein. Ein möglicher Satz in einem Beschreibungstext wäre

> „Then *the bunny stands up on his hindlegs.“*

aus Text E.7. des Text-Korpus.

Sätze wie

> *„The bunny and the frog jump one after the other.“*

können zum Einen durch ein *do in order* verwirklicht werden, das für beide Objekte die Methode *jump* enthält, und zum Anderen durch ein *for all in order*, in dessen Liste sich die Objekte *bunny* und *frog* befinden, wobei hier, wie bereits erwähnt, eine Regel festgelegt werden sollte, wann welches Konstrukt sinnvoller ist.

Natürlich muss bei *do in order* auch beachtet werden, dass der Nutzer nicht immer eindeutige Schlüsselwörter verwenden wird, sondern für eine Aktion, die einer anderen folgt, einfach einen neuen Satz beginnt. Außerdem kann man sich nicht ausnahmslos darauf verlassen, dass durch eines der oben genannten Wörter ein *Do in order* impliziert wird, schließlich ist die normale Abfolge von Methoden, die sich nicht innerhalb einer Kontrollstruktur befinden, ebenfalls nacheinander.

Eine Möglichkeit wäre hier, nur dann ein *Do in order* einzubauen, wenn man sich bereits in einem *Do together* befindet.

4.2.6. Do Together

Wie in Kapitel Abschnitt 2.1 bereits erwähnt, können mit Hilfe eines *Do together*-Konstrukts mehrere Methoden gleichzeitig ausgeführt werden. Ein Beispiel aus der Musterlösung des Text-Korpus ist:

> *„The bunny's head turns forward by a small amount* while *the frog turns to face the bunny* at the same time.*“*

Wie man sieht ist „at the same time“ ein wichtiges Schlüsselwort, aber auch „while“ weist auf eine Gleichzeitigkeit hin. Hierbei sollte man jedoch beachten, dass das Wort „while“ auch auf eine while-Schleife hinweisen kann. Natürlich lässt sich so eine Nebenläufigkeit auch allein durch die Wortfolge „at the same time“ ausdrücken, wie in Beschreibung E.1. des Text-Korpus:

> *„The bunny looks in the mailbox and* at the same time *the frog turns to face the bunny.*

Weitere Schlüsselwörter wären *simultaneously, concurrently, at one time, meanwhile, as long as, together.* Schwieriger wird es, wenn solche eindeutigen Begriffe weggelassen werden und zwei gleichzeitig auszuführende Aktionen nur mit „and“ verbunden sind. In diesem Fall kann man nicht eindeutig sagen, ob es sich um eine Nebenläufigkeit oder eine nacheinander ausgeführte Abfolge von Aktionen handelt. So zum Beispiel in *„The dog walks and barks.“*.

Auch kommt es vor, dass sich eine Gleichzeitigkeit über mehrere Sätze erstreckt.In einem solchen Fall ist es hilfreich, wenn der darauffolgende Satz mit einem Wort wie „then“ beginnt.

Adverb	n
twice doubly	2
triply trebly thrice	3

Tabelle 4.2.: Adverbien zum Ausdrücken von „n-mal“ .

4.3. Loop

Wie bereits in Kapitel Abschnitt 2.1 erwähnt, entspricht *Loop* in Alice einer gewöhnlichen for-Schleife. Eine solche wird in Alice beschrieben durch die Anzahl der Ausführungen und beinhaltet alle Methoden(-abfolgen), die entsprechend oft ausgeführt werden sollen.

Im Skript zur Filmsequenz *Bunny* finden sich hauptsächlich weniger komplizierte Schleifen wie zum Beispiel:

```
1 Loop 3 times times
2         bunny.hop
```

Diese wird in den Beschreibungstexten immer mit dem (Teil-)Satz *„The bunny hops 3 times.“* ausgedrückt. Die Form *n times*, zu deutsch als *n-mal* mit einer natürlichen Zahl n, die größer ist als eins, ist also eine beliebte Art, eine Schleife in einem natürlichsprachlichen Text auszudrücken. Die Zahl kann hierbei auch ausgeschrieben vorkommen.

Bei folgender Schleife aus dem Textkorpus von [Ham12] zeigt sich in den zugehörigen Beschreibungstexten außer *two times* auch noch die Möglichkeit *twice*.

```
1 Loop 2 times times
2         frog.hop
```

Aus diesem Grund müssen in der Implementierung nicht nur Beschreibungen der Form *n times* für die Anzahl der Schleifenausführungen behandelt werden, sondern insbesondere auch Adverbien wie *twice, thrice, etc.* Zu diesem Zweck wurde zuerst recherchiert, welche derartigen besonderen Begriffe es für den Ausdruck „n-mal“ gibt. Es zeigte sich, dass es solche – abgesehen von „einmal“ – nur für „zwei-“ und „dreimal“ gibt (siehe Tabelle 4.2). Eine dritte Möglichkeit, die jedoch weniger erwünscht ist, jedoch trotzdem beachtet werden soll, ist eine ungenaue Aussage wie *a few times*.

Um die Kontrollstruktur *Loop* zu realisieren, wurde zum einen festgelegt, wie eine *Loop*-Annotation später aussehen soll (siehe Unterabschnitt 4.3.1) und außerdem analysiert, wie eine Annotation des Stanford Parsers von Sätzen, die eine Schleife implizieren, aussieht und wie man diese nutzen kann, um eine Schleife zu erkennen.

4.3.1. Wie soll eine Loop-Annotation aussehen?

Um später den Zugriff auf die Annotation zu erleichtern, ist es sinnvoll, dass jede Kontrollstruktur ihren eigenen Annotationstyp hat. Sein Name gleicht hierbei dem der Kontrollstruktur, in diesem Fall also *Loop*.

Außerdem galt es, eine Entscheidung zu treffen, wo die Annotation stehen soll. Eine erste Überlegung war, sie um das Verb beziehungsweise die Methode zu schreiben. Nun ist es in Alice jedoch so, dass ein Methodenname nicht immer nur aus einem Verb besteht, sondern oftmals aus mehreren Wörtern. So gibt es zum Beispiel die Methode *turn*, aber auch die Methode *turn to face*. Die Aufgabe, Satzteile mit Methoden zu verknüpfen ist jedoch

nicht Teil dieser Arbeit, sondern wird in der Diplomarbeit von Sebastian Weigelt [Wei14] behandelt. Es ist also nicht sinnvoll, im Rahmen der vorliegenden Arbeit herauszufinden, was im Satz alles zu einer Methode gehört. Des Weiteren können innerhalb einer Schleife mehrere Methoden ausgeführt werden. Würde man nun alle diesen Methoden zugeordneten Verben beziehungsweise Satzteile mit einer Annotation versehen, bestünde die Gefahr, dass später bei der Erzeugung des Skripts mehrere gleichartige Schleifen mit jeweils einer Methode erstellt würden.

Es ist also ratsam, die Annotation nicht einem Verb zuzuordnen, sondern besser den Wörtern, die eine Schleife und ihre Anzahl an Durchläufen definieren. Jedoch ist es dann zwingend notwendig, auf das entsprechende Verb zu verweisen. Bei „turn to face“ beispielsweise sollte also auf das Wort „turn“ verwiesen werden.

Folgende Informationen sollte eine Loop-Annotation also unbedingt beinhalten:

- Die Anzahl der Schleifendurchläufe und
- das Verb, auf das sich die Schleife bezieht.

Diese werden durch die Attribute *LoopCycles* und *Verb* angegeben.

In manchen Fällen kann es unklar sein, ob sich eine Schleife auf mehrere Verben eines Satzes bezieht oder nur auf eines. So zum Beispiel in folgendem Satz:

„The bunny hops and nods two times.“

Hier stellte sich die Frage, ob das Kaninchen nun einmal hüpft und dann zweimal nickt, oder ob es sowohl zweimal hüpft als auch nickt. Dieser Satz ist mehrdeutig und somit nicht eindeutig einzuordnen. Es galt also eine Regel festzusetzen, wie hier unterschieden werden soll. Da es einfacher ist auszudrücken, dass nur eines der Verben zur Schleife gehört, indem man zum Beispiel ein „then“ einfügt wie im Beispiel

„The bunny hops and then nods two times“,

wurde entschieden, dass sich eine Schleife bei einer Konjunktion mehrerer Verben immer auf alle diese Verben bezieht, sofern nicht explizit angegeben ist, dass sie sich nur auf das Verb unmittelbar vor dem Schleifenausdruck beziehen soll. Folgende Regel soll dies nochmals zusammenfassen:

Regel 4.3.1.1 *Bei einer Verbalphrase mit einer Konjunktion mehrerer Verben und einem Schleifenausdruck der Form*

Verb 1, Verb 2, ... and Verb n loop,

wobei loop *für einen Schleifenausdruck steht, umschließt die Schleife alle Verben, es sei denn, sie sind explizit getrennt durch Begriffe wie „then“. Das heißt, es gilt*

Loop{Verb 1, Verb 2, ...,Verb n}

4.3.2. Schleifenerkennung mit Hilfe des Stanford Parsers

Um die entsprechenden auf eine Schleife hinweisenden Begriffe zu erkennen, bedient sich diese Arbeit des Stanford Parsers (siehe Abschnitt 2.4). Auf einen zu annotierenden Beschreibungstext wird also zuerst der Stanford Parser angewendet. Dieser annotiert, wie bereits erläutert, Satz- und Wortarten und ist so eine große Hilfe bei der Suche nach bestimmten Wörtern. Es musste also zuerst analysiert werden, wie die Parser-Annotationen der Schlüsselwörter aussehen.

```
<SyntaxTreeNode label="VP" sentenceID="1">
    <SyntaxTreeNode label="VBZ" sentenceID="1">
        <Dependency index="3" sentenceID="1" typedDependencies="
            root; dobj~5; nsubj~2" typedDependenciesCCprocessed="
            dobj~5; nsubj~2" typedDependenciesCollapsed="dobj~5;
            nsubj~2">
             hops
        </Dependency>
    </SyntaxTreeNode>
    <SyntaxTreeNode label="NP" sentenceID="1">
        <SyntaxTreeNode label="CD" sentenceID="1">
            <Dependency index="4" sentenceID="1">
                 three
            </Dependency>
        </SyntaxTreeNode>
        <SyntaxTreeNode label="NNS" sentenceID="1">
            <Dependency index="5" sentenceID="1"
                typedDependencies="num~4"
                typedDependenciesCCprocessed="num~4"
                typedDependenciesCollapsed="num~4">
                 times
            </Dependency>
        </SyntaxTreeNode>
    </SyntaxTreeNode>
</SyntaxTreeNode>
```

Quelltextausschnitt 4.1: Ausschnitt des mit dem Stanford Parser in GoldenGATE annotierten Satzes „The bunny hops three times.“

Quelltextausschnitt 4.1 zeigt eine solche Annotation am Beispiel des Satzes „The bunny hops three times“, wobei hier zugunsten der Übersicht nur der Teil *hops three times* angegeben ist. Wie man sieht ist dieser Teil umschlossen von einer *SyntaxTreeNode*-Annotation mit dem Label-Attribut *VP*, was einen Verbalphrase markiert. Der Satzteil „three times“ wiederum wird als *NP* annotiert, was für eine Nominalphrase steht. Eine Stufe tiefer hat „three“ den Part-Of-Speech-Tag *CD*, wird also als Kardinalzahl erkannt, und „times“ mit *NNS* als Substantiv im Plural. Da „one time“ keine Schleife mit sich bringt, genügt es auch, „times“ nur im Plural zu betrachten.

Betrachtet man das Verb „hops“ genauer, fällt auf, dass dieses den Part-Of-Speech-Tag *VBZ* hat, wodurch es als Verb in der dritten Person Singular bezeichnet wird. Es besitzt innerhalb seiner *Dependency*-Annotation auch einen Verweis auf das Objekt „times“ über dessen Index, hier *dobj~5*. Diese Eigenschaften sollen später auch bei der Implementierung genutzt werden, um Satzteile der Form „n times“ zu erkennen. Für Satzteile dieser Form ließen sich auch keine Unregelmäßigkeiten bei der Annotation durch den Stanford Parser feststellen, weshalb im Weiteren davon ausgegangen wird, dass diese immer korrekt erfolgt.

Bei den Adverbien ist dies jedoch leider nicht der Fall. Hier wurde beobachtet, dass der Stanford Parser diese Begriffe oftmals nicht als Adverb, sondern als irgendeine andere Wortart erkennt. Würde man diesen Fehler übergehen, hätte das zur Folge, dass viele Schleifen in Beschreibungstexten gar nicht erkannt würden. Es ist also wünschenswert, diese Begriffe weitgehend trotzdem zu erkennen. Quelltextausschnitt 4.2 zeigt nun den mit dem Stanford Parser annotierten Textteil „hops twice“ aus „The bunny hops three

```
<SyntaxTreeNode label="VP" sentenceID="1">
    <SyntaxTreeNode label="VBZ" sentenceID="1">
        <Dependency index="3" sentenceID="1" typedDependencies="
            root; nsubj~2; advmod~4" typedDependenciesCCprocessed=
            "nsubj~2; advmod~4" typedDependenciesCollapsed="nsubj
            ~2; advmod~4">
             hops
        </Dependency>
    </SyntaxTreeNode>
    <SyntaxTreeNode label="RB" sentenceID="1">
        <SyntaxTreeNode label="ADVP" sentenceID="1">
            <Dependency index="4" sentenceID="1">
                 twice
            </Dependency>
        </SyntaxTreeNode>
    </SyntaxTreeNode>
</SyntaxTreeNode>
```

Quelltextausschnitt 4.2: Ausschnitt des mit dem Stanford Parser in GoldenGATE annotierten Satzes „The bunny hops twice.“

times.“ Wenn alles richtig annotiert wird, erhält „twice“ wie hier die Wortartmarkierung *RB* für Adverb und die Satzteilmarkierung *ADVP*, die eine Adverbialphrase angibt. Auch hier besitzt das Verb „hops“ wieder eine Abhängigkeit vom Adverb, im Gegensatz zu einem Objekt wie bei obigem Fall, ist das hier *advmod~4*.

Die geplante Vorgehensweise ist nun also folgende: Mit Hilfe der Annotationen des Stanford Parsers sollen alle Begriffe, die auf eine Schleife hinweisen, gesucht sowie die Indizes der entsprechenden Verben ermittelt werden. Diese sollen am Ende zusammen mit der Anzahl der Schleifendurchläufe als Attribute einer Loop-Annotation um die gefundenen Begriffe angegeben werden.

5. Implementierung

Nachdem das Konzept in Kapitel 4 entworfen und vorgestellt wurde, folgt nun die Implementierung. Im Rahmen der vorliegenden Arbeit wurde ein GoldenGATE-Plugin für das Erkennen von Kontrollstrukturen so weit implementiert, dass Schleifen erkannt und annotiert werden können. Dieses Plugin kann entsprechend der Prioritätenliste (siehe Tabelle 4.1) nach und nach um die restlichen Kontrollstrukturen erweitert werden. So wird nicht für jede einzelne Kontrollstruktur ein eigenes GoldenGATE-Plugin benötigt.

5.1. Grundaufbau des GoldenGATE-Plugins

Wie bereits erwähnt soll es ein einziges Plugin für die Erkennung und Annotation von Kontrollstrukturen geben. *Kontrollstrukturen* bildet die Hauptklasse des Plugins. Als solche muss sie die Gamta-Schnittstelle *de.uka.ipd.idaho.gamta.util.Analyzer* implementieren und ist eine Unterklasse von *de.uka.ipd.idaho.gamta.util.AbstractConfigurableAnalyzer*. Somit überschreibt sie die als Hauptmethode dienende Methode *process*, die den zu annotierenden Text in Form einer *MutableAnnotation* namens *doc* übergeben bekommt. Dieser sollte in der Regel bereits mittels Stanford Parser annotiert worden sein. In dieser Methode wird zum Beispiel die Methode *searchForLoops* aus der Klasse *Loop* aufgerufen (siehe Abschnitt 5.2). Abbildung 5.1 veranschaulicht die Zusammenhänge der verschiedenen Klassen.

5.2. Implementierung der Loop-Annotation

Die Hauptfunktionalität der Loop-Annotation ist in der Klasse *Loop* angesiedelt, die im Konstruktor ebenfalls *doc* übergeben bekommt. Die Klasse *Loop* wiederum besteht hauptsächlich aus der Methode *searchForLoops*, in der mit Hilfe der Parser-Annotationen nach Begriffen gesucht wird, die auf eine Schleife hinweisen (siehe Abschnitt A) und in der diese dann eine Loop-Annotation erhalten. Zu diesem Zweck bedient sie sich einiger Hilfsmethoden.

5.2.1. Schleifenerkennung mittels Stanford Parser

Wie bereits in Unterabschnitt 4.3.2 erläutert, befinden sich die gesuchten Schlüsselwörter immer innerhalb einer Verbalphrase. Da es möglich ist, dass mehrere gleichartige Satzteile ineinander verschachtelt sind und es so zu einer mehrfachen Annotation derselben Schleife

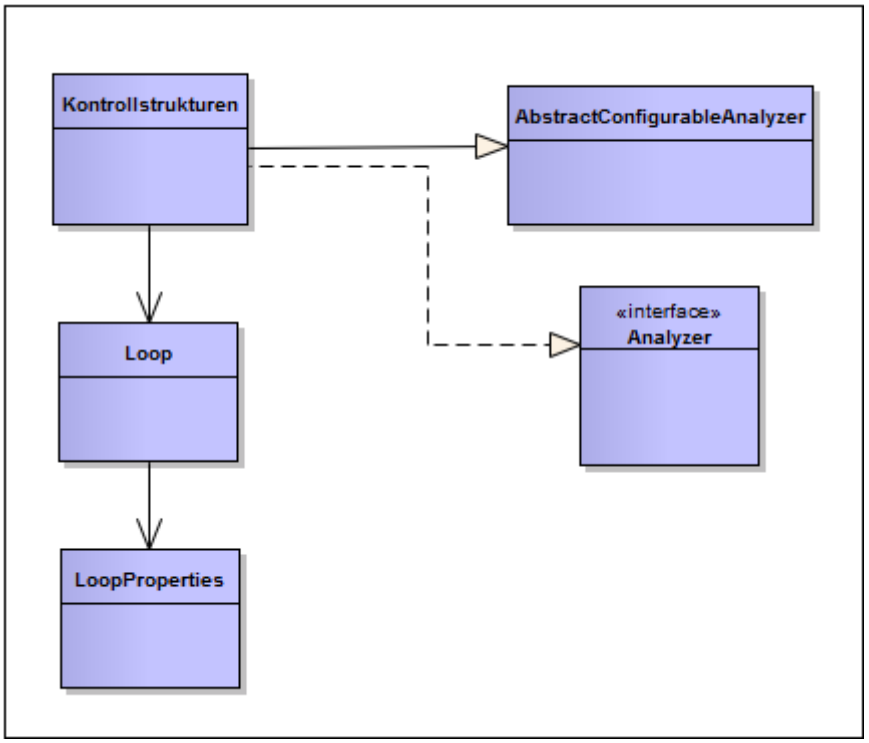

Abbildung 5.1.: Klassendiagramm des Plugins

kommen könnte, werden zu Beginn mittels GPath (siehe Abschnitt 2.3) nur alle inneren Verbalphrasen und dann alle darin enthaltenen Verben ermittelt. Diese werden dann für jedes Verb zum Einen durchsucht nach Schleifenbegriffen mit „times“ (Hilfsmethode *searchForNTimes*) und zum Anderen nach solchen wie „twice“ (Hilfsmethode *adverbialLoop*).

5.2.1.1. Schleifenbeschreibungen der Form „n times“

Bei Schleifenbeschreibungen der Form „n times“ wird zuerst nach dem Wort „times“ gesucht. Hierzu wird innerhalb der als Verbalphrase annotierten Satzteile nach einer annotierten Nominalphrase, das heißt mittels GPath nach einer SyntaxTreeNode-Annotation mit dem Label *NP*, gesucht. Auch hierbei werden wieder nur die inneren Nominalphrasen überprüft, um doppelte Annotationen zu vermeiden. Innerhalb dieser wiederum wird unter allen als Substantiv Plural annotierten Wörter das Wort „times“ gesucht.

Anschließend muss noch untersucht werden, ob das Wort vor „times“ eine Zahl ist. Diese sind im Normalfall mit *CD*, also als Kardinalzahl, annotiert, jedoch wurde festgestellt, dass der Stanford Parser sie oftmals auch als Adjektiv erkennt und mit *JJ* versieht. Daher werden beide Möglichkeiten in Betracht gezogen. Hierzu werden nun zuerst alle Kardinalzahlen und alle Adjektive innerhalb der Nominalphrase daraufhin überprüft, ob sie zum Einen direkt vor „times“ stehen und zum Anderen, ob es sich tatsächlich um eine Zahl handelt.

Das Problem bei der Erkennung und Umwandlung der als String vorliegenden natürlichen Zahlen besteht darin, dass sie nicht nur in numerischer sondern auch in ausgeschriebener Form engegeben sein können. Diejenigen in numerischer Form können einfach in einen Integer umgewandelt werden, wodurch auch gleich überprüft wird, ob es sich wirklich um natürliche Zahlen handelt. Sind die Zahlen jedoch ausgeschrieben, funktioniert das Umwandeln in einen Integer nicht. Es musste also eine andere Möglichkeit gefunden werden, diese in Zahlen numerischer Form umzuwandeln. Hier kam nun WordNet zum Einsatz.

Wie sich herausstellte, wird in den Synsets bei WordNet bei Eingabe einer ausgeschriebenen Zahl die zugehörige numerische Form ausgegeben. Allerdings gelingt dies nur bei Zahlen bis hundert zuverlässig. Da sich der Verwendungszweck der Zahlen in dieser Arbeit jedoch auf die Angabe von Schleifendurchläufen in Beschreibungstexten für Videos beschränkt, kann dieses Problem vernachlässigt und somit davon ausgegangen werden, dass der Benutzer keine Zahlen größer als Hundert benötigt.

Ist ein Umwandeln der gefundenen Zahl von String zu Integer also nicht möglich, wird in WordNet nach der entsprechenden numerischen Form angefragt. Bekommt man auch hier kein Ergebnis, wurde entweder eine Zahl größer als Hundert eingegeben oder das erkannte Wort war keine Zahl.

Für den Fall, dass keine Zahl vor „times“ gefunden wird, wird überprüft, ob es sich um eine ungenaue Angabe wie zum Beispiel „a few times“ handelt. In beiden Fällen ist die Schleife erkannt und muss noch annotiert werden. Wie dies vonstattengeht wird in Unterabschnitt 5.2.2 erläutert.

5.2.1.2. Schleifenbeschreibungen durch Adverbien

Da die Adverbien, die eine Schleife ausdrücken können (siehe auch Tabelle 4.2), vom Stanford Parser oftmals falsch annotiert werden, wird hierbei nicht nur für die Adverbien sondern für alle Wörter innerhalb der Verbalphrase überprüft, ob es sich um ein Adverb handelt, das eine Schleife impliziert. Hierzu werden alle Wörter dieser Verbalphrase mit den Adverbien aus Tabelle 4.2 verglichen und gegebenenfalls die zugehörige Zahl zwischengespeichert.

5.2.2. Annotieren der Schleifen-Begriffe

Wenn nun die Anzahl an Schleifendurchläufen sowie die Begriffe, die eine Loop-Annotation erhalten sollen, ermittelt wurden, wird nun noch der Index des Verbes, beziehungsweise gegebenenfalls mehrerer Verben, benötigt. Mit der Hilfsmethode *getVerbIndex* und dem Index des Schleifenbegriffs wird dann nach einem *typedDependencies*-Eintrag beim Verbs gesucht, der eine Abhängigkeit zwischen diesem und dem Schleifenbegriff angibt. Wurde ein solcher gefunden, wird der Index des Verbs in das Attribut *Verb* in der Annotation *Loop* geschrieben.

Wird keine solche Abhängigkeit gefunden, wird im Fall mehrerer Verben in der Verbalphrase über die *Dependeny*-Annotationen nach einer Konjunktion zwischen diesen und gegebenenfalls nach einer entsprechenden Abhängigkeit zwischen einem dieser Verben und dem Schleifenbegriff gesucht.

Um alle für eine Schleife benötigten Informationen zurückgeben zu können, wurde die Klasse *LoopProperties* angelegt, deren Instanzen Datenstrukturen für eben diese Informationen sind. So enthalten sie den Start- und den Endindex des Satzteiles, der annotiert werden soll sowie die Anzahl an Schleifendurchläufen und die Indizes der Verben, auf die sich die Schleife bezieht.

Über verlinkte Listen solcher *LoopProperties*-Objekte werden diese Informationen an die Methode *annotateLoopWords* weitergegeben, in der die Annotation dann, sofern es sie noch nicht gibt, vorgenommen wird.

5.2.2.1. Noch nicht erkannte Konjunktionen von Verben

Noch immer kann es Fälle geben, in denen nicht alle Verben, auf die sich eine Schleife bezieht, erkannt wurden. Da aufgefallen ist, dass die Annotation gut funktioniert, wenn eine Trennung der Verben durch Wörter wie „then“ im Satz gegeben ist, wie zum Beispiel

in „The bunny hops, nods and then blinks twice“, wobei sich „twice“ hier laut Regel 4.3.1.1 auf die Verben „hops“ und „nods“ bezieht, jedoch weniger gut, wenn dieses „then“ nicht enthalten ist, wird, nachdem alle Annotationen erfolgt sind, noch einmal überprüft, ob in Sätzen ohne Wörter wie „then“ noch weitere Verben gefunden werden, die bei der Auflistung der Indizes in der *Loop*-Annotation übergangen wurden. Diese werden der Annotation dann gegebenenfalls noch hinzugefügt.

Ein Beispiel für einen annotierten Satz, der eine Schleife beschreibt, ist in Unterunterabschnitt C.2.16 des Anhangs angegeben.

6. Evaluation

Zur Evaluation wird das Plugin mit den Texten aus dem Textkorpus aus [Ham12] getestet und die Ergebnisse hiervon in Abschnitt 6.1 ausgewertet. Da im Textkorpus jedoch nur wenige Kontrollstrukturen, insbesondere Schleifen, vorkommen, wurde mit Alice eine weitere kurze Videosequenz „Cowboy“ erstellt, deren Fokus auf der Verwendung von Schleifen liegt, und diese anschließend von mehreren Personen beschrieben. Hierzu werden die Ergebnisse in Abschnitt 6.2 ausgewertet.

In diesem Kapitel werden zugunsten einer besseren Übersichtlichkeit nur Besonderheiten und Probleme erfasst. Die kompletten annotierten Texte finden sich jedoch in Abschnitt C des Anhangs für den Textkorpus sowie in Unterabschnitt D.3 für die Beschreibungstexte zum Video „Cowboy“.

6.1. Evaluation mit dem Textkorpus zum Video „Bunny“

Wie mit Hilfe des Quelltextes der Hauptmethode zur Filmsequenz „Bunny“ (siehe Unterabschnitt C.1) nachvollzogen werden kann, enthält dieses Video fünf Schleifen, die mit dem in dieser Arbeit entwickelten Plugin erkannt und annotiert werden sollen. Einige der Personen, die das Video beschrieben haben, haben jedoch nicht beachtet, wie oft etwas getan wird, weshalb die Schleifen in diesen Fällen auch nicht erkannt werden konnten (siehe z.B. Unterunterabschnitt C.2.3, C.2.6, C.2.12, C.2.13). Hierbei ist zu beachten, dass dies später kein Problem mehr sein wird, da der Benutzer dann weiß, wie oft etwas geschehen soll und dies dann auch angeben wird.

Des Weiteren ist zu erwähnen, dass es diverse Fälle gibt, in denen die Verb-Attribute der Loop-Annotation nicht ganz mit dem übereinstimmen, was wirklich im Quelltext zu „Bunny“ steht. Ob diese Annotation dann im Rahmen der Evaluation als korrekt angesehen wird, hängt davon ab, ob die in dieser Arbeit festgelegten Regeln korrekt umgesetzt wurden. So wird beispielsweise in Beschreibungstext 3 (siehe Unterunterabschnitt C.2.4) der Satz „Bunny turns a bit right to face the mailbox and hops 3 times to left part of the screen and sits down in front of the mailbox“ folgendermaßen annotiert:

```
Bunny turns a bit right to face the mailbox and hops
<Loop LoopCycles="3" Verb="2; 11">
3 times
</Loop>
to left part of the screen and sits down in front of the mailbox.
```

Wie man sieht, wird nach Regel 4.3.1.1 so annotiert, dass sich die Schleife auf „turns“ und „hops“ bezieht, was nicht dem eigentlichen Quelltext von „Bunny“ entspricht. Dies liegt an der bereits in Unterabschnitt 4.3.1 erläuterten Mehrdeutigkeit dieses Satzes, in dem nicht klar zu sagen ist, ob sich „3 times“ nur auf „hops“ beziehen soll oder auf beide Verben. Nach Regel 4.3.1.1 wurde jedoch festgelegt, dass bei solchen mehrdeutigen Fällen davon ausgegangen wird, dass es sich auf beide Verben bezieht. In obigem Satz sollte der Nutzer diese Mehrdeutigkeit also beispielsweise durch Einfügen von „then“ nach „and“ auflösen.

Vollständig korrekt annotiert wurden Beschreibungstext 1 und die Musterlösung, jedoch sind viele Fehler in den anderen Texten auf falsche Annotationen durch den Stanford Parser zurückzuführen.

Im Folgenden werden nun die Fehler und Probleme beim Annotieren der Beschreibungstexte sortiert nach ihrer Art aufgeführt.

6.1.1. Fehler aufgrund des Stanford Parser

Des Öfteren trat bei der Evaluation der Texte das Problem auf, dass eine Schleife nicht erkannt werden konnte, weil eine Annotation des Stanford Parsers nicht korrekt erfolgte, was dann jedoch kein Fehler dieser Arbeit ist.

Ein Beispiel hierfür findet sich in Beschreibungstext 3 (sieh Unterunterabschnitt C.2.4):

```
Bunny turns right and taps
<Loop LoopCycles="2">
two times
</Loop>
with left foot.
```

Hier wurde überhaupt kein Verb als Bezugswort für die Schleife erkannt. Betrachtet man nun die Annotationen des Stanford Parsers zeigt sich, dass das Wort „taps“ fälschlicherweise als *NNS* annotiert wurde. „Turns“ hingegen wurde zwar als Verb erkannt, jedoch findet sich in seiner *Dependency*-Annotation keine Abhängigkeit zum Wort „times“, was auch der Grund dafür ist, dass es nicht erkannt wurde. Wie nun mit Hilfe eines Beispiels mit korrekter Annotation herausgefunden werden konnte, wäre dies anders, wenn „taps“ korrekt als Verb annotiert worden wäre. Dann würde man bei ihm eine Abhängigkeit auf „times“ finden und es gäbe eine Konjunktion zwischen beiden Verben, was dazu führen würde, dass auch die Schleife korrekt erkannt würde. Daraus lässt sich schließen, dass dies kein Fehler des entwickelten Plugins ist.

Auch in Beschreibungstext 4 wird eine Schleife nicht erkannt. Hier wurde das Wort „taps“ in dem Satz „The Bunny turns to face the Mailbox and taps its foot two times“ mit *VB* annotiert, was der Grundform eines Verbs entspricht. Offensichtlich handelt es sich hierbei jedoch um ein Verb in der dritten Person Singular. Grundformen von Verben wurden in der Implementierung jedoch nicht beachtet, da Verben innerhalb einer Schleife immer in konjugierter Form auftreten. Somit konnte auch hier keine Schleife erkannt werden. Das selbe Problem trat auch bei Beschreibungsttext 14 auf.

Ein etwas schwieriger einzuschätzender Fall ist ein Satz aus Beschreibungstext 6, der wie folgt annotiert wurde:

```
Then, Bunny turns left to Frog and waves to Frog
<Loop LoopCycles="2" Verb="11">
twice
</Loop>
with the left paw.
```

Wie zu sehen ist, wurde hier als Verb, auf das sich die Schleife bezieht, das Wort „Frog“ erkannt. Auch das liegt an einer falschen Annotation von Seiten des Stanford Parsers, der dieses Wort als Verb erkannt hat. Nach Regel 4.3.1.1 sollte sich diese Schleife jedoch auf zwei Verben einer Konjunktion („turns“ und „waves“) beziehen. „waves“ wurde vom Stanford Parser nicht als Verb, sondern als Substantiv Plural erkannt, „turns“ hingegen wurde als Verb erkannt. Des Weiteren wurde auch „left“ wieder als Verb erkannt. Es wird also vermutet, dass der Parser „turns left“ gemeinsam angesehen hat, wie es zum Beispiel bei „has left“ auch richtig wäre. Dadurch, dass jedoch so vieles falsch interpretiert wurde, gibt es den *Dependency*-Annotationen nach zu urteilen keine Konjunktion zwischen den Verben. Somit konnte das entwickelte Plugin auch nur das als Verb annotierte „Frog“ erkennen.

Auch in der dritten Schleife in Beschreibungstext 9 sowie in der ersten Schleife in Beschreibungstext 3 wurde „left“ als Verb erkannt, jedoch nicht ins *Loop*-Attribut *Verb* aufgenommen. Nach einem Test des gleichen Satzes, jedoch mit „left“ ersetzt durch „right“, in dem alles korrekt annotiert wurde, kann gesagt werden, dass auch dieser Fehler auf der fehlerhaften Parser-Annotation basiert.

In Beschreibungstext 14 zeigt die dritte Schleife erneut, dass das entwickelte Plugin Regel 4.3.1.1 nicht beachten kann, wenn nirgends eine *typedDependencies*-Eintrag gefunden wird, der auf eine Konjunktion mehrerer Verben hinweist.

6.1.2. Fehler des Plugins

Nicht beachtete Ausdrucksweisen für Schleifen

Einen weiteren Fall, in dem eine Schleife nicht erkannt wird, zeigen die folgenden Beispiele aus Beschreibungstext 3 und 6:

> *„Frog turns and makes 2 hops back to mailbox.“*

> *„Bunny hops with 3 leaps to Broccoli and eats it.“*

Hierbei wird das Verb „hops“ umschrieben durch „makes 2 hops“ beziehungsweise „hops with 3 leaps“. Eine derartige Schreibweise wurde bei der Implementierung des Plugins nicht beachtet und somit auch nicht erkannt. Solche Ausdrucksweisen treten im Textkorpus insgesamt fünf mal auf: einmal in Beschreibungstext 3 und je zweimal in den Beschreibungstexten 6 und 7.

Fehler aufgrund nicht behandelter *Dependency*-Annotationen

Der folgende Ausschnitt aus Beschreibungstext 9 zeigt eine weitere Art von Fehler, der auf einer *Dependency*-Annotation des Parsers basiert, die bei den während der Implementierung verwendeten Beispieltexten nicht aufgetreten ist. Bei der Implementierung wurde davon ausgegangen, dass es immer ein *typedDependencies*-Attribut beim Verb gibt, das auf das Wort „times“ verweist. In diesem Fall besitzt „hops“ jedoch keinen solchen *typedDependencies*-Eintrag, sondern stattdessen einen, der auf das Wort „back“ verweist, das wiederum den Verweis auf „times“ besitzt. Bei derartigen Abhängigkeiten wird also das zugehörige Verb zu einer Schleife vom entwickelten Plugin nicht erkannt.

```
The frog hops
<Loop LoopCycles="2">
two times
</Loop>
back to the bunny.
```

	#Schleifen	Erkennung			Parsefehler
		korrekt	unvollständig	keine	
Musterlösung	5	5	0	0	0
Text 1	5	5	0	0	0
Text 2	0	0	0	0	0
Text 3	5	4	0	1	2
Text 4	1	1	0	0	1
Text 5	0	0	0	0	0
Text 6	3	1	0	2	1
Text 7	4	2	0	2	0
Text 8	1	1	0	0	0
Text 9	5	4	1	0	1
Text 10	2	1	1	0	0
Text 11	0	0	0	0	0
Text 12	0	0	0	0	0
Text 13	1	1	0	0	0
Text 14	5	5	0	0	2
GESAMT	**37**	**30**	**5**	**2**	**7**

Tabelle 6.1.: Verteilung der korrekt, unvollständig und nicht erkannten Schleifen der Beschreibungstexte von „Bunny“

Der selbe Fehler tritt auch bei der ersten Schleife in Beschreibungstext 10 auf.

Um diesen Fehler zu beheben, können für den Fall, dass eine gesuchte Abhängigkeit beim Verb nicht gefunden werden kann, die *Dependency*-Annotationen anderer von ihm abhängiger Wörter überprüft werden.

6.1.3. Auswertung

Um das Ergebnis der Auswertung der Evaluation anhand des Textkorpus nicht zu verfälschen, werden hierbei nur Schleifen beachtet, die im Text auch erwähnt wurden sowie nur diejenigen, bei denen kein Fehler durch fehlerhafte Stanford Parser Annotationen aufgetreten ist.

Wie sich zeigte wurden in den Texten insgesamt 37 Schleifen beschrieben, wobei sieben davon eine fehlerhafte Annotation durch den Parser erhielten. Somit bleiben 30 Schleifen zur Auswertung. Hiervon wurden 23 Stück korrekt, fünf gar nicht und zwei nicht vollständig annotiert. Die fünf, die gar nicht annotiert wurden, waren diejenigen, die durch eine Umschreibung wie „makes 3 hops“ ausgedrückt wurden. Die beiden, die nicht vollständig annotiert wurden, sind diejenigen, die eine nicht beachtete Art von *Dependency*-Annotationen enthielten. Es wurden also $76,\bar{6}\%$ der beschriebenen Schleifen korrekt annotiert. Die genauen Ergebnisse zeigt Tabelle 6.1.

6.2. Evaluation mit den Beschreibungen zum Video „Cowboy“

Die im Rahmen dieser Bachelorarbeit mit Alice erstellte Videosequenz „Cowboy“, deren Quelltext in Unterabschnitt D.1 im Anhang angegeben ist, enthält drei Schleifen. Im Folgenden wird nun, wie zuvor auch schon mit dem Textkorpus, mit Hilfe der Musterlösung zum Video sowie dreier Beschreibungstexte verschiedener Personen ausgewertet, wie gut die beschriebenen Schleifen mit dem entwickelten Plugin erkannt und annotiert werden.

Die Musterlösung konnte hierbei vollständig und korrekt annotiert werden. Auch hier werden die Fehler im Folgenden sortiert nach ihrer Art aufgeführt.

6.2.1. Fehler aufgrund des Stanford Parser

In Beschreibungstext 1 wurde die folgende Schleifenbeschreibung nicht erkannt:

> *„Both the cowboy and the camel nod four times.“*

Schaut man sich dazu die Annotationen des Stanford Parsers an, sieht man, dass „nod“ statt als Verb als Substantiv erkannt wurde, wodurch auch das entwickelte Plugin die zugehörige Schleife nicht erkennen konnte.

6.2.2. Fehler des Plugins

Wie auch schon beim Textkorpus aufgefallen und in Unterabschnitt 6.1.2 beschrieben, wurden in der Implementierung nicht alle möglichen Arten von Abhängigkeiten beachtet, da nicht klar war, dass diese auftreten können. So wurde auch bei Beschreibungstext 2 von „Cowboy“ eine Schleife nicht vollständig annotiert:

```
Then the cowboy and the camel nod for
<Loop LoopCycles=”4”>
four times
</Loop>
.
```

Hier fehlt das *Verb*-Attribut mit dem Index von „nods“. Das liegt daran, dass die Abhängigkeit des Worts „times“ nicht im *typedDependencies*-Attribut von „nods“, sondern in dem von „for“ steht. Die Ursache der unvollständigen Annotation liegt hier also auf Seiten des entwickelten Plugins.

Das selbe Problem wurde auch bei der zweiten Schleife des dritten Beschreibungstextes angetroffen.

6.2.3. Schleifenbeschreibung durch „again“

In Beschreibungstext 2 wird eine Schleife auf eine Art beschreiben, die in den bisherigen Texten noch nie auftrat:

> *„The cowboy jumps up an kick his left leg. Comes to the ground an jumps in the air again an again.“*

Dieser Satz soll eine Schleife mit drei Durchläufen ausdrücken, in Wirklichkeit jedoch bedeutet dieses „again and again“, dass das Springen immer wieder ausgeführt wird. Es handelt sich hierbei also um eine ungenaue Ausdrucksweise. Beachtet man, dass es letztlich nicht darum geht, ein gesehenes Video zu beschreiben, sondern darum, auszudrücken was in einem zu erzeugenden Video geschehen soll, ist es unwahrscheinlich, dass der Benutzer eine Schleife auf obige Weise beschreibt. Der Benutzer überlegt sich schließlich im Vorfeld genau, wie oft etwas passieren soll und drückt es dann auch explizit aus. Daher wird diese Schleifenbeschreibung in der Auswertung weder als falsch noch als korrekt annotiert angesehen, sondern stattdessen außer Acht gelassen.

	#Schleifen	Erkennung			Parsefehler
		korrekt	unvollständig	keine	
Musterlösung	3	3	0	0	0
Text 1	3	2	0	0	1
Text 2	1	0	1	0	0
Text 3	3	2	1	0	0
GESAMT	**10**	**7**	**2**	**0**	**1**

Tabelle 6.2.: Verteilung der korrekt, unvollständig und nicht erkannten Schleifen der Beschreibungstexte von „Cowboy“

6.2.4. Auswertung

Wie auch schon bei der Auswertung des Plugins bezüglich des Textkorpus, wird auch hier von allen beschriebenen Schleifen nur diejenigen betrachtet, die vom Stanford Parser korrekte Annotationen erhielten. Des weiteren wird, wie bereits in Unterabschnitt 6.2.3 gesagt, die durch Aneinanderreihung des Worts „again“ ausgedrückte Schleife außer Acht gelassen. Somit bleiben es neun beschriebene Schleifen. Von diesen wurden sieben korrekt und zwei unvollständig erkannt. Das ergibt eine Erkennungsrate von $77,\bar{7}\%$, ein ähnliches Ergebnis wie auch schon bei der Auswertung mit dem Textkorpus (siehe Unterabschnitt 6.1.3). Auch dieses Ergebnis wird noch einmal in einer Tabelle veranschaulicht Tabelle 6.2.

6.3. Zusammenfassung beider Ergebnisse

Fasst man die Ergebnisse der vorhergehenden Abschnitte zusammen, wurden von 39 beschriebenen Schleifen 30 korrekt erkannt, während vier unvollständig und fünf überhaupt nicht erkannt wurden. Das macht wiederum eine gemeinsame Erkennungsrate von 76,9% sowie 10,3% unvollständige Erkennung und 12,8%, in denen die Schleife überhaupt nicht erkannt wurde.

Die Unvollständigkeit einiger Annotationen lässt sich darauf zurückführen, dass das Plugin Abhängigkeitsangaben zu Schleifenausdrücken nur bei Verben findet, diese sich jedoch, wie sich herausstellte, vor allem bei längeren Sätzen oftmals auch bei anderen Wörtern wie Präpositionen stehen. Dies ist ein Problem des Plugins, das in Zukunft durchaus behoben werden kann.

Ein anderes Problem sind Umschreibungen von Verben mit zugehörigen Substantiven wie in „makes 3 hops“ für das Verb „hop“. Diese Möglichkeit eine Schleife auszudrücken, wurde in diesem Plugin bisher nicht implementiert. Es ist jedoch denkbar, in Zukunft auch derartige Beschreibungen mit einzubeziehen, wozu Untersuchungen angestellt werden sollten, welche Möglichkeiten es gibt, Verben auf obige Weise zu umschreiben.

7. Zusammenfassung und Ausblick

Natürlichsprachliche Programmierung ist ein vielversprechender Ansatz, um besonders Nichtinformatikern den Einstieg in das für sie oftmals abschreckende Gebiet der Programmierung zu erleichtern und in ihnen das Interesse daran zu wecken.

Das Projekt *Programmieren in natürlicher Sprache* soll es nun ermöglichen, durch Anfertigen einer Art Drehbuch in natürlicher Sprache eine Videosequenz für die Rahmenarchitektur Alice zu erzeugen. So soll der Anwender intuitiv beschreiben können, was in einer solchen Videosequenz geschehen soll, und der hiebei erstellte Text anschließend in für Alice verständlichen Quelltext zur Erstellung dieser Videosequenz umgewandelt werden. Als besonders an Programmieranfänger gerichtete Rahmenarchitektur ist Alice mit seiner realitätsnahen Namensgebung und seinem einfachen Aufbau eine optimale Basis für dieses Projekt.

Das Projekt ist aufgeteilt in mehrere Teilgebiete, wobei sich die vorliegende Arbeit mit der Erkennung und Annotation von Alice-Kontrollstrukturen beschäftigte. So wurden sämtliche Kontrollstrukturen, die Alice bietet, analysiert und Schlüsselwörter gesammelt, die in natürlichsprachlichen Texten dazu verwendet werden können, die entsprechenden Kontrollstrukturen zu beschreiben. Dann wurde ein Plugin für GoldenGATE entwickelt, das diese Kontrollstrukturen in natürlichsprachlichen Texten erkennen und ihre Schlüsselwörter annotieren soll. Bislang ist dieses in der Lage, in den Texten ausgedrückte Schleifen zu annotieren.

Bei der Evaluation anhand der Beschreibungstexte aus dem „Bunny“-Textkorpus sowie derer, die zu der in dieser Arbeit mit Alice erstellten Videosequenz „Cowboy“ gesammelt wurden, konnte eine Schleifenerkennungsrate von 76,9% erzielt werden. Weitere 10,3% wurden teilweise erkannt. Es zeigte sich, dass die Erkennungsrate sehr stark von der Formulierung abhängt, gerade bei kurzen Sätzen werden sehr gute Ergebnisse erzielt, wohingegen lange und verschachtelte Sätze schwieriger zu erkennen sind. Nach dieser Evaluation ist nun klar, worauf ein Benutzer achten sollte,um bestmögliche Ergebnisse zu erzielen. Eine grafische Benutzeroberfläche im späteren Programm, die den Benutzer auf ungenaue Angaben hinweist und ihm die Möglichkeit gibt, diese zu konkretisieren, wäre hierfür von großem Nutzen. Durch Erweiterung des Plugins um die Erkennung von Schleifenausdrücken wie „The bunny *makes 2 hops*“ sowie eine Korrektur der Erkennung der Abhängigkeiten erkannter Schlüsselwörter kann die Erkennungsrate in Zukunft außerdem stark erhöht werden.

Der nächste Schritt wird nun sein, das entwickelte Plugin um die noch fehlenden Kontrollstrukturen zu erweitern, wobei auf den dazu bisher gesammelten Erkenntnissen aufgebaut werden kann. Im Hinblick auf das Gesamtprojekt sind bereits weitere Teilgebiete in Arbeit, wie zum Beispiel der Aufbau der Anfangsszenerie, die Verknüpfung der Wörter im Text mit den Entitäten aus Alice sowie die Erkennung von Methodenparametern. Im Anschluss daran können diese und eventuell folgende Komponenten dann zu einem Gesamtprodukt vereint werden, das Beschreibungstexte, die mit den in dieser und anderen Arbeiten entwickelten Plugins annotiert werden, in ausführbaren Quelltext zur Erzeugung von Videosequenzen in Alice umwandelt.

Literaturverzeichnis

[CAB+00] Conway, Matthew ; Audia, Steve ; Burnette, Tommy ; Cosgrove, Dennis ; Christiansen, Kevin: Alice: lessons learned from building a 3D system for novices. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA : ACM, 2000 (CHI '00). – ISBN 1–58113–216–6, 486–493

[Con97] Conway, Matthew J.: *Alice: Easy-to-Learn 3D Scripting for Novices*, Faculty of the School of Engineering and Applied Science, University of Virginia, Diss., Dezember 1997. `http://alice.org/publications/ConwayDissertation.PDF`

[Gul12] Gulwani, Sumit: Synthesis From Examples: Interaction Models and Algorithms. In: *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2012

[Ham12] Hampel, Sina: *Programmieren in natürlicher Sprache: Aufbau eines Alice-Korpus*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Bachelor's Thesis, 2012. `https://svn.ipd.kit.edu/trac/AliceNLP/wiki/Theses/hampel_ba`

[Hey13] Hey, Tobias: *Programmieren in natürlicher Sprache: Zeitliches Sortieren*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Bachelor's Thesis, Dezember 2013. `https://svn.ipd.kit.edu/trac/AliceNLP/wiki/Theses/kohlmann_ba`

[KM03] Klein, Dan ; Manning, Christopher D.: Fast Exact Inference with a Factored mModel for Natural Language Parsing. In: *Advances in Neural Information Processing Systems 15*. Cambridge, MA, USA : MIT Press, 2003 (NIPS 2002), 3–10

[LL05] Liu, Hugo ; Lieberman, Henry: Metafor: Visualizing Stories as Code. In: *IUI '05: Proceedings of the 10th International Conference on Intelligent User Interfaces*. New York, NY, USA : ACM, 2005. – ISBN 1–58113–894–6, 305–307

[LM06] Little, Greg ; Miller, Robert C.: Translating keyword commands into executable code. In: *Proceedings of the 19th annual ACM symposium on User interface software and technology*. New York, NY, USA : ACM, 2006 (UIST '06). – ISBN 1–59593–313–1, 135–144

[MMS93] Marcus, Mitchell P. ; Marcinkiewicz, Mary A. ; Santorini, Beatrice: Building a large annotated corpus of English: the penn treebank. In: *Comput. Linguist.* 19 (1993), Juni, Nr. 2, 313–330. `http://dl.acm.org/citation.cfm?id=972470.972475`. – ISSN 0891–2017

[Pet12] Peters, Oleg: *Programmieren in natürlicher Sprache: Aufbau einer Alice-Ontologie*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Bachelor's Thesis, 2012. `https://svn.ipd.kit.edu/trac/AliceNLP/wiki/Theses/peters_ba`

[PRZH00] Price, David ; Rilofff, Ellen ; Zachary, Joseph ; Harvey, Brandon: NaturalJava: A natural language interface for programming in Java. In: *Proceedings of the 5th international conference on Intelligent user interfaces* ACM, 2000, S. 207–211

[SBPT07] Sautter, Guido ; Böhm, Klemens ; Padberg, Frank ; Tichy, Walter: Empirical evaluation of semi-automated XML annotation of text documents with the GoldenGATE editor. In: *Proceedings of the 11th European conference on Research and Advanced Technology for Digital Libraries*. Berlin, Heidelberg : Springer-Verlag, 2007 (ECDL'07). – ISBN 3–540–74850–4, 978–3–540–74850–2, 357–367

[Sek14] Seker, Ali: *Programmieren in natürlicher Sprache - Erfassung von Methodenargumenten aus natürlicher Sprache*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Diplomarbeit, 2014. `https://svn.ipd.kit.edu/trac/AliceNLP/wiki/Theses/seker_da`

[Wei12] Weigelt, Sebastian: *Programmieren in natürlicher Sprache: Aufbau einer Alice-Ontologie – Korpus-Ontologie-Assoziation –*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Studienarbeit, 2012. `https://svn.ipd.kit.edu/trac/AliceNLP/wiki/Theses/weigelt_sa`

[Wei14] Weigelt, Sebastian: *Programmieren in natürlicher Sprache: Erkennung und semantische Assoziation von Entitäten in natürlichsprachlichen Texten*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Diplomarbeit, Februar 2014. `https://svn.ipd.kit.edu/trac/AliceNLP/wiki/Theses/weigelt_da`

Anhang

A. Sammlung möglicher Schlüsselwörter

Kontrollstruktur	Schlüsselwörter
if/else	if, in [...] case, provided (that) else, elsewise, otherwise
while	while, as long as, provided (that), until
for all in order	in order, consecutively, one by one, one at a time, one after another, one after the other, in succession, successive(ly), then, afterwards, after this/that, later, next, subsequently, finally, in the end
for all together	at once, simultaneous(ly), coeval(ly), coexistent, coincidental, coinstantaneous, concomitant, concurrent(ly), contemporaneous(ly), at the same time, synchronistic(ally), synchronous, while, as long as,
do in order	in order, consecutively, one by one, one at a time, one after another, one after the other, in succession, successive(ly)
do together	at once, simultaneous(ly), coeval(ly), coexistent, coincidental, coinstantaneous, concomitant, concurrent(ly), contemporaneous(ly), at the same time, synchronistic(ally), synchronous, while, as long as
loop	a few times, n times (n=Zahl grøßer 1), twice, doubly, thrice, trebly, triply

B. Penn-Tagset

B.1. Wortartmarkierungen

CC	Coordinating conjunction
CD	Cardinal number
CT	Determiner
EX	Existential *there*
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper non, plural
PDT	Predeterminer
POS	Possessive ending
PRP	Personal pronoun
PRP$	Possessive pronoun
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Particle
SYM	Symbol
TO	*to*
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or present participle
VBN	Verb, past participle
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present
WDT	Wh-determiner
WP	Wh-pronoun
WP$	Possessive wh-pronoun
WRB	Wh-adverb

B.2. Satzteilmarkierungen

ADJP	Adjective Phrase
ADVP	Adverb Phrase
CONJP	Conjunction Phrase
FRAG	Fragment
INTJ	Interjection
LST	List marker
NAC	Not A Constituent
NP	Noun Phrase
NX	Used within certain complex noun phrases to mark the head of the noun phrase.
PP	Prepositional Phrase
PRN	Parenthetical
PRT	Particle
QP	Quantifier Phrase
RRC	Reduced Relative Clause
UCP	Unlike Coordinated Phrase
VP	Verb Phrase
WHADJP	*wh*-adjective Phrase
WHADVP	*wh*-adverb Phrase
WHNP	*wh*-noun Phrase
WHPP	*wh*-prepositional Phrase
X	Unknown, uncertain or unbracketable

C. Textkorpus „Bunny“

Im Folgenden sind die Hauptmethode des Alice-Quelltext zu „Bunny“ aus [Ham12] sowie die Beschreibungstexte inklusive *Loop*-Annotation angegeben.

C.1. Hauptmethode des Alice-Quelltext zur Videosequenz „Bunny“

```
No variables
Wait 1 second
bunny turn to face broccoli
Loop 3 times times
    bunny.hop
Wait 0,5 seconds
bunny.eat food = broccoli
Wait 0,5 seconds
bunny turn to face frog
Loop 2 times times
    bunny.tapfoot foot = left
frog.ribbit
frog turn left 0,25 revolutions
Loop 3 times times
    frog.hop
bunny turn to face mailbox
Loop 3 times times
    bunny.hop
bunny.openmailbox
Do together
    bunny.upperBody.head turn forward 0.15 revolutions duration =
        0,5 seconds
    frog turn to face bunny
Loop 2 times times
    frog.hop
frog.ribbit
```

C.2. Bunny-Beschreibungstexte mit annotierten Schleifen

Zugunsten einer besseren Übersichtlichkeit werden die Beschreibungstexte aus dem Textkorpus im Folgenden nur annotiert mit der *Loop*-Annotation angegeben, jedoch ohne die Annotationen des Stanford Parsers. Anschließend jedoch wird Satz 6 aus der Musterlösung vollständig annotiert als Beispiel für einen korrekt annotierten Satz (siehe Unterunterabschnitt C.2.16)sowie Satz 9 aus Beschreibungstext 6 als Beispiel für eine unvollständige Loop-Annotation aufgrund fehlerhafter Stanford Parser Annotationen (siehe Unterunterabschnitt C.2.17) angegeben.

C.2.1. Musterlösung

```
The ground is covered with grass. In the background there is a
    palm tree on the left. In the foreground there is a frog on
    the left facing east -southeast and a broccoli on the right.
    Behind the broccoli there is a bunny facing south -southwes t.
     Behind the frog, to the left, there is a mailbox facing
    southeast. The bunny turns to face the broccoli. The bunny
    hops
<Loop LoopCycles="3" Verb="3">
three times
</Loop>
. A very short time passes. The bunny eats the broccoli. A very
    short time passes. The bunny turns to face the frog. The bunny
     taps its left foot
<Loop LoopCycles="2" Verb="3">
twice
</Loop>
. The frog ribbits. The frog turns left by 90 degrees. The frog
    hops
<Loop LoopCycles="3" Verb="3">
three times
</Loop>
. The bunny turns to face the mailbox. The bunny hops
<Loop LoopCycles="3" Verb="3">
three times
</Loop>
. The bunny opens the mailbox. The bunny 's head turns forward by
     a small amount while the frog turns to face the bunny at the
    same time. The frog hops
<Loop LoopCycles="2" Verb="3">
twice
</Loop>
. The frog ribbits.
```

C.2.2. Beschreibungstext 1

```
The ground is covered with grass, the sky is blue. In the
    background on the left hand side there is a PalmTree. In the
    foreground on the left hand side there is a closed Mailbox
    facing southeast. Right to the mailbox there is a Frog facing
    east. In the foreground on the right hand side there is a
    Bunny facing southwest. In front of the Bunny there is a
    Broccoli. The Bunny turns to face the Broccoli. The Bunny hops
<Loop LoopCycles="3" Verb="3">
three times
</Loop>
to the Broccoli. The Bunny eats the Broccoli. The Bunny turns to
    face the Frog. The Bunny taps his foot
<Loop LoopCycles="2" Verb="3">
twice
</Loop>
. The Frog ribbits. The Frog turns to face northeast. The frog
    hops
```

```
<Loop LoopCycles="3" Verb="3">
three times
</Loop>
to northeast. The Bunny turns to face the Mailbox. The Bunny hops
<Loop LoopCycles="3" Verb="3">
three times
</Loop>
to the Mailbox. The Bunny opens the Mailbox. The Bunny looks in
   the Mailbox and at the same time the Frog turns to face the
   Bunny. The Frog hops
<Loop LoopCycles="2" Verb="3">
two times
</Loop>
to the Bunny. The Frog ribbits.
```

C.2.3. Beschreibungstext 2

```
I see a palm tree on the left of the screen, a mailbox in front
   of it. In the foreground there sits a frog on the left and a
   hare on the right of the screen. The hare hops and bends
   forward to eat something. The frog looks at what the hare does
   . The hare turns to the frog and taps its hindpaw. The frog
   turns its back to the observer and hops to the back (into the
   background). The hare hops to the mailbox and opens it. He
   looks into the mailbox. The frog turns back to the hare and
   also hops to the mailbox.
```

C.2.4. Beschreibungstext 3

```
Ground is covered with green grass. Left background with palm.
   Left in front of palm: mailbox in brown american style. Left
   in front of mailbox: frog in green colour. Right front: bunny
   in white colour. Half of screen is blue sky. In front of bunny
    on right side: green broccoli. Bunny turns left and hops
<Loop LoopCycles="3" Verb="5">
3 times
</Loop>
in direction of viewer of video. Bunny eats broccoli. Bunny turns
    right and taps
<Loop LoopCycles="2">
two times
</Loop>
with left foot. Frog says something to bunny. Frog hops
<Loop LoopCycles="3" Verb="2">
3 times
</Loop>
from front left part of the scene to centre of the screen. Frog
   sits with back to viewer. Bunny turns a bit right to face the
   mailbox and hops
<Loop LoopCycles="3" Verb="2; 11">
3 times
</Loop>
to left part of the screen and sits down in front of the mailbox.
    Bunny opens mailbox with right hand. Bunny looks into the
```

```
    mailbox. Frog turns and makes 2 hops back to mailbox. Frog
    sits behind bunny and looks into his direction. Frog says
    something to bunny.
```

C.2.5. Beschreibungstext 4

```
The floor is covered in grass. In the far right foreground is a
    Broccoli. In the far left is a Mailbox and in front of it is a
     Frog. Behind the Mailbox in the background is a PalmTree.
    Behind the Broccoli in the background is a Bunny. The Bunny is
     facing the south, the Frog is facing the Broccoli. The Bunny
    hops toward the Broccoli and eats it. The Bunny turns to face
    the Mailbox and taps its foot two times. The Frog ribbits and
    hops into the background. The Bunny hops toward the Mailbox
    and opens the Mailbox. The Frog hops toward the Bunny.
```

C.2.6. Beschreibungstext 5

```
The ground is covered with grass. In the background on the left
    there is a PalmTree. In the foreground in front of the
    PalmTree is a Mailbox. On the right side of the Mailbox there
    is a frog. On the right side in the foreground there is a
    Broccoli. Behind the Broccoli in the Background is a Bunny.
    The Bunny hops to the broccoli. The bunny eats the Broccoli
    and turns left to the frog. The bunny taps its foot. The frog
    ribbits. The frog turns to the Palm tree. The Frog hops in the
     background. The Bunny hops to the Mailbox. The Bunny opens
    the Mailbox. The frog turns to the Bunny. The frog hops to the
     Bunny.
```

C.2.7. Beschreibungstext 6

```
The ground is covered with grass. The sky is blue. In the
    background there is a PalmTree on the left side. Left of the
    palm tree there is a mailbox. On the right side of the palm
    tree there sits a frog. In the background on the left there is
     Bunny. In front of Bunny there is Broccoli. Bunny hops with 3
     leaps to Broccoli and eats it. Then, Bunny turns left to Frog
     and waves to Frog
<Loop LoopCycles="2" Verb="11">
twice
</Loop>
with the left paw. Frog opens the mouth and closes it again. Frog
     turns backward and hops with three leaps to the mailbox,
    opens it and looks into the mailbox. The frog turns around and
     hops back to Bunny and the open mailbox.
```

C.2.8. Beschreibungstext 7

```
On the upper half of the screen is a clear blue sky. On the lower
     half of the screen is a green meadow. On the right in the
    foreground sits a Bunny on his hind legs facing southwest.
    Before the Bunny stands a mushroom. On the left side in the
    foreground stands a palm with a brown trunk and green leaves.
```

```
    Before the palm and somewhat to the left of it stands a shut
    mailbox on a brown post. To the right of the mailbox a little
    more in the foreground sits a green frog with a yellow belly
    facing southeast in the direction of the mushroom. The Bunny
    turns his head and body slightly to the left facing exactly to
     the south. The Bunny jumps upward
<Loop LoopCycles="3" Verb="3">
three times
</Loop>
and then bends forward, lies down on the meadow and eats the
    mushroom. Then the Bunny stands up on his hind legs again and
    turns right facing west in the direction of the mailbox. The
    Bunny taps his left foot leg
<Loop LoopCycles="2" Verb="3">
twice
</Loop>
on the ground. Then the frog turns to the left facing northeast
    and making three jumps to the background. The Bunny hops to
    the mailbox, opens it and looks into it. The frog turns to the
     right now facing southwest and hops in two jumps to the right
     side of the Bunny.
```

C.2.9. Beschreibungstext 8

```
The ground is covered with grass. The sky is blue. In the
    background there is a palm tree on the left. In the foreground
     there is a broccoli on the right. Left to the broccoli there
    is a frog facing the broccoli. Behind the frog, to the left,
    there is a mail box facing southeast. Behind the Broccoli
    there is a rabbit facing south. The bunny turns to face the
    broccoli. The bunny hops towards the broccoli. The bunny eats
    the broccoli. The bunny turns to face the frog. The bunny taps
     its left foot
<Loop LoopCycles="2" Verb="3">
two times
</Loop>
. The frog ribbits. The frog turns to face north. The frog hops
    to the background. The bunny turns to the mail box. The bunny
    hops towards the mailbox. The bunny opens the mail box. The
    bunny turns its head down. Meanwhile the frog quickly turns to
     face the bunny. The frog hops towards the bunny. The frog
    ribbits. There is nothing happening for 3 seconds.
```

C.2.10. Beschreibungstext 9

```
The ground is covered with grass. The sky is blue. There is a
    palmtree on the left side in the background. On the right side
     there is a big white bunny facing southwest. In front of
    palmtree there is a mailbox. Right to the mailbox, there is a
    frog facing southeast. In front of the bunny, there is a
    broccoli. The bunny turns left to the south and hops
<Loop LoopCycles="3" Verb="3; 9">
three times
```

```
</Loop>
to the broccoli. The bunny eats the broccoli. The bunny turns
   right to face the frog. The bunny taps
<Loop LoopCycles="2" Verb="3">
two times
</Loop>
on the ground. The frog ribbits. The frog turns left and hops
<Loop LoopCycles="3" Verb="6">
three times
</Loop>
to the northeast. The bunny hops
<Loop LoopCycles="3" Verb="3">
three times
</Loop>
to the mailbox. The bunny opens the mailbox door. The bunny looks
    into the mailbox. The frog turns right to face the bunny. The
    frog hops
<Loop LoopCycles="2">
two times
</Loop>
back to the bunny. The frog ribbits.
```

C.2.11. Beschreibungstext 10

```
There is a bunny in the picture turning towards the camera. A
   piece of Brokkoli is in front of him. A frog sits left of the
   Brokkoli facing it. Behind the frog there is a mailbox. Behind
    the mailbox there is a palm. Everything is placed on grass.
   The bunny hops to the Brokkoli and eats it. Then it turns and
   moves to the rabbit and taps its foot. The frog turns left and
    hops
<Loop LoopCycles="3">
3 times
</Loop>
away. The bunny then hops
<Loop LoopCycles="3" Verb="4">
3 times
</Loop>
to the mailbox, opens it and looks down on the floor. While
   looking down, the frog turns around, hops to the bunny and
   ribbits.
```

C.2.12. Beschreibungstext 11

```
The ground is covered with grass. There is a palm on the left
   side of the ground, before the palm there is a mailbox. In
   front of the mailbox, there is a frog. On the right side of
   the ground, there is a broccoli. A rabbit is sitting behind
   the broccoli. The rabbit hops, moving toward the broccoli. The
    rabbit eats the broccoli. The rabbit turns to the frog and
   taps foot. The frog ribbits, turns left and hops to the back.
   The bunny hops to the mailbox, opens it, and looks into the
   mailbox. Meanwhile, the frog turns to the bunny, moves forward
   , and ribbits.
```

C.2.13. Beschreibungstext 12

```
The ground is covered with grass. In the foreground to the right,
    there is a Broccoli. Behind the Broccoli, there is a Bunny
   facing south. In the foreground there is a Frog on the left
   facing southeast. Behind the frog, to the left, is a Mailbox.
   Behind the Mailbox, to the right, is a PalmTree. The Bunny
   hops to the Broccoli. It eats the Broccoli. The Bunny turns
   towards the frog. A very short time passes. The Bunny taps its
    foot. The Frog ribbits, turns to face north and hops away
   from the mailbox. A very short time passes. The Bunny hops to
   the Mailbox. The Bunny opens the mailbox. It is empty. The
   Bunny plays a sound (disappointment). A very short time passes
   . The frog turns around to face the bunny. The Frog hops
   towards the Bunny. A very short time passes. The Frog ribbits.
```

C.2.14. Beschreibungstext 13

```
The ground is covered with grass. In the background there is a
   palmtree on the far left. In front of the palmtree there is a
   mailbox. In the foreground to the right there is a broccoli.
   Right of the mailbox there is a frog facing the broccoli. On
   the right side behind the broccoli there is a bunny facing the
    frog. The bunny faces the broccoli and hops to it. Then the
   bunny eats the broccoli and turns to face the frog. The bunny
   taps its foot
<Loop LoopCycles="2" Verb="3">
twice
</Loop>
. The frog ribbits, turns a half revelation to the left and hops
   away from mailbox. The bunny hops to the mailbox and opens
   mailbox. The frog turns to face the bunny, hops to the bunny
   and ribbits once.
```

C.2.15. Beschreibungstext 14

```
The floor is covered with grass. In the background on the left,
   there is a palm tree. In front of the palm tree, there is a
   postal mailbox facing southeast. In the foreground on the left
   , there sits a green frog with a yellow belly facing
   eastsoutheast. In the background on the right, there sits a
   white bunny facing southsouthwest. In south of the bunny,
   there is a broccoli. The bunny turns to the broccoli and hops
<Loop LoopCycles="3" Verb="3; 8">
3 times
</Loop>
towards it. The bunny bends down and eats the broccoli. The bunny
    sets pose, turns face to face to the frog and taps his left
   foot twice. The frog ribbits once, then turns north and hops
<Loop LoopCycles="3" Verb="10">
3 times
</Loop>
. The bunny turns to the mailbox and hops
<Loop LoopCycles="3" Verb="3; 8">
```

```
3 times
</Loop>
. The bunny opens the mailbox 's door with his left hand, lowers
    his head and looks into the mailbox. The frog turns around and
    hops
<Loop LoopCycles="2" Verb="3; 6">
twice
</Loop>
towards the bunny.
```

C.2.16. Vollständige, korrekte Annotation von Satz 6 aus der Musterlösung

```
<SyntaxTreeNode label="S" sentenceID="6">
  <SyntaxTreeNode label="ROOT" sentenceID="6">
    <SyntaxTreeNode label="NP" sentenceID="6">
      <SyntaxTreeNode label="DT" sentenceID="6">
        <Dependency index="1" sentenceID="6">
The
        </Dependency>
      </SyntaxTreeNode>
      <SyntaxTreeNode label="NN" sentenceID="6">
        <Dependency index="2" sentenceID="6" typedDependencies="
            det~1" typedDependenciesCCprocessed="det~1"
            typedDependenciesCollapsed="det~1">
bunny
        </Dependency>
      </SyntaxTreeNode>
    </SyntaxTreeNode>
    <SyntaxTreeNode label="VP" sentenceID="6">
      <SyntaxTreeNode label="VBZ" sentenceID="6">
        <Dependency index="3" sentenceID="6" typedDependencies="
            root; dobj~5; nsubj~2" typedDependenciesCCprocessed="
            dobj~5; nsubj~2" typedDependenciesCollapsed="dobj~5;
            nsubj~2">
hops
        </Dependency>
      </SyntaxTreeNode>
      <SyntaxTreeNode label="NP" sentenceID="6">
        <Loop LoopCycles="3" Verb="3">
          <SyntaxTreeNode label="CD" sentenceID="6">
            <Dependency index="4" sentenceID="6">
three
            </Dependency>
          </SyntaxTreeNode>
          <SyntaxTreeNode label="NNS" sentenceID="6">
            <Dependency index="5" sentenceID="6"
                typedDependencies="num~4"
                typedDependenciesCCprocessed="num~4"
                typedDependenciesCollapsed="num~4">
times
            </Dependency>
          </SyntaxTreeNode>
        </Loop>
      </SyntaxTreeNode>
    </SyntaxTreeNode>
    <SyntaxTreeNode label="." sentenceID="6">
.
    </SyntaxTreeNode>
  </SyntaxTreeNode>
</SyntaxTreeNode>
```

C.2.17. Vollständige Annotation von Satz 9 aus Beschreibungstext 6 mit fehlerhafter Stanford Parser Annotation

```
<SyntaxTreeNode label="S" sentenceID="9">
  <SyntaxTreeNode label="ROOT" sentenceID="9">
    <SyntaxTreeNode label="RB" sentenceID="9">
      <SyntaxTreeNode label="ADVP" sentenceID="9">
        <Dependency index="1" sentenceID="9">
Then
        </Dependency>
      </SyntaxTreeNode>
    </SyntaxTreeNode>
    <SyntaxTreeNode label="," sentenceID="9">
,
    </SyntaxTreeNode>
    <SyntaxTreeNode label="NNP" sentenceID="9">
      <SyntaxTreeNode label="NP" sentenceID="9">
        <Dependency index="3" sentenceID="9">
Bunny
        </Dependency>
      </SyntaxTreeNode>
    </SyntaxTreeNode>
    <SyntaxTreeNode label="VP" sentenceID="9">
      <SyntaxTreeNode label="VBZ" sentenceID="9">
        <Dependency index="4" sentenceID="9" typedDependencies="
            root; nsubj~3; dep~5; advmod~1"
            typedDependenciesCCprocessed="nsubj~3; dep~5; advmod~1
            " typedDependenciesCollapsed="nsubj~3; dep~5; advmod~1
            ">
turns
        </Dependency>
      </SyntaxTreeNode>
      <SyntaxTreeNode label="VP" sentenceID="9">
        <SyntaxTreeNode label="VBN" sentenceID="9">
          <Dependency index="5" sentenceID="9" typedDependencies=
              "prep~6; prep~10" typedDependenciesCCprocessed="
              prep_to~9; prepc_to~11; prep_to~7"
              typedDependenciesCollapsed="prepc_to~11; prep_to~7"
              typedDependenciesKilled="KILL~10; KILL~6">
left
          </Dependency>
        </SyntaxTreeNode>
        <SyntaxTreeNode label="PP" sentenceID="9">
          <SyntaxTreeNode label="TO" sentenceID="9">
            <Dependency index="6" sentenceID="9"
                typedDependencies="pobj~7" typedDependenciesKilled
                ="KILL~7">
to
            </Dependency>
          </SyntaxTreeNode>
          <SyntaxTreeNode label="NP" sentenceID="9">
            <SyntaxTreeNode label="NNP" sentenceID="9">
              <SyntaxTreeNode label="NP" sentenceID="9">
                <Dependency index="7" sentenceID="9"
                    typedDependencies="cc~8; conj~9"
```

```
                    typedDependenciesCCprocessed="conj_and~9"
                    typedDependenciesCollapsed="conj_and~9">
Frog
                  </Dependency>
                </SyntaxTreeNode>
              </SyntaxTreeNode>
              <SyntaxTreeNode label="CC" sentenceID="9">
                <Dependency index="8" sentenceID="9">
and
                </Dependency>
              </SyntaxTreeNode>
              <SyntaxTreeNode label="NNS" sentenceID="9">
                <SyntaxTreeNode label="NP" sentenceID="9">
                  <Dependency index="9" sentenceID="9">
waves
                  </Dependency>
                </SyntaxTreeNode>
              </SyntaxTreeNode>
            </SyntaxTreeNode>
          </SyntaxTreeNode>
          <SyntaxTreeNode label="PP" sentenceID="9">
            <SyntaxTreeNode label="TO" sentenceID="9">
              <Dependency index="10" sentenceID="9"
                  typedDependencies="pcomp~11"
                  typedDependenciesKilled="KILL~11">
to
              </Dependency>
            </SyntaxTreeNode>
            <SyntaxTreeNode label="VP" sentenceID="9">
              <SyntaxTreeNode label="S" sentenceID="9">
                <SyntaxTreeNode label="VBG" sentenceID="9">
                  <Dependency index="11" sentenceID="9"
                      typedDependencies="advmod~12; prep~13"
                      typedDependenciesCCprocessed="prep_with~16;
                      advmod~12" typedDependenciesCollapsed="
                      prep_with~16; advmod~12"
                      typedDependenciesKilled="KILL~13">
Frog
                  </Dependency>
                </SyntaxTreeNode>
                <SyntaxTreeNode label="RB" sentenceID="9">
                  <SyntaxTreeNode label="ADVP" sentenceID="9">
                    <Dependency index="12" sentenceID="9">
                      <Loop LoopCycles="2" Verb="11">
twice
                      </Loop>
                    </Dependency>
                  </SyntaxTreeNode>
                </SyntaxTreeNode>
                <SyntaxTreeNode label="PP" sentenceID="9">
                  <SyntaxTreeNode label="IN" sentenceID="9">
```

```
                    <Dependency index="13" sentenceID="9"
                        typedDependencies="pobj~16"
                        typedDependenciesKilled="KILL~16">
with
                    </Dependency>
                  </SyntaxTreeNode>
                  <SyntaxTreeNode label="NP" sentenceID="9">
                    <SyntaxTreeNode label="DT" sentenceID="9">
                      <Dependency index="14" sentenceID="9">
the
                      </Dependency>
                    </SyntaxTreeNode>
                    <SyntaxTreeNode label="JJ" sentenceID="9">
                      <Dependency index="15" sentenceID="9">
left
                      </Dependency>
                    </SyntaxTreeNode>
                    <SyntaxTreeNode label="NN" sentenceID="9">
                      <Dependency index="16" sentenceID="9"
                          typedDependencies="amod~15; det~14"
                          typedDependenciesCCprocessed="amod~15; det
                          ~14" typedDependenciesCollapsed="amod~15;
                          det~14">
paw
                      </Dependency>
                    </SyntaxTreeNode>
                  </SyntaxTreeNode>
                </SyntaxTreeNode>
              </SyntaxTreeNode>
            </SyntaxTreeNode>
          </SyntaxTreeNode>
        </SyntaxTreeNode>
      </SyntaxTreeNode>
      <SyntaxTreeNode label="." sentenceID="9">
.
      </SyntaxTreeNode>
    </SyntaxTreeNode>
  </SyntaxTreeNode>
</SyntaxTreeNode>
```

D. Videosequenz „Cowboy“

Im Folgenden sind weitere Informationen zur Videosequenz „Cowboy“ gegeben.

D.1. Alice-Quelltext zur Videosequenz „Cowboy“

D.1.1. Hauptmethode *world.my first method()*

```
No variables
Do together
    Loop 2 times times
        camel.nod
    cowboy.walk howFar = 2
cowboy turn right 0.25 revolutions
Loop 3 times times
    Do together
        cowboy.kick
        cowboy.clapHands
Loop 4 times times
    Do together
        cowboy.nod
        camel.nod
```

D.1.2. Methode *camel.nod()*

```
No variables
camel.neck turn forward 0.25 revolutions duration = 0.75 seconds
camel.neck turn backward 0.25 revolutions duration = 0.75 seconds
\end{listing}

\subsubsection{Methode \emph{cowboy.clapHands()}}
\begin{lstlisting}
No variables
Do together
    cowboy.Chest.jacket.RArm.RForArm turn backward 0.12 revolutions
    cowboy.Chest.jacket.Larm.LForArm turn backward 0.12 revolutions
    cowboy.Chest.jacket.Larm.LForArm turn right 0.27 revolutions
    cowboy.Chest.jacket.RArm.RForArm turn left 0.27 revolutions
    cowboy.Chest.jacket.Larm.LForArm.LHand turn left 0.25 revolutions
    cowboy.Chest.jacket.RArm.RForArm.RHand turn right 0.25 revolutions
Do together
    cowboy.Chest.jacket.Larm.LForArm turn forward 0.12 revolutions
    cowboy.Chest.jacket.RArm.RForArm turn forward 0.12 revolutions
    cowboy.Chest.jacket.Larm.LForArm turn left 0.27 revolutions
    cowboy.Chest.jacket.RArm.RForArm turn right 0.27 revolutions
    cowboy.Chest.jacket.Larm.LForArm.LHand turn right 0.25 revolutions
    cowboy.Chest.jacket.RArm.RForArm.RHand turn left 0.25 revolutions
```

D.1.3. Methode *cowboy.kick()*

```
No variables
Do together
    cowboy move up 1 meter duration = 1 second
    Do in order
        Do together
            cowboy.RLeg turn backward 0.25 revolutions duration = 0.5
                seconds
            cowboy.RLeg.Rshin turn forward 0.25 revolutions duration =
                0.5 seconds
```

```
            cowboy.LLeg.LShin turn forward 0.25 revolutions duration =
                0.5 seconds
        cowboy.RLeg.Rshin turn backward 0.25 revolutions duration = 0.5
            seconds
Do together
    cowboy move down 1 meter
    cowboy.RLeg turn forward 0.25 revolutions
    cowboy.RLeg.Rshin turn backward 0.25 revolutions
    cowboy.LLeg.LShin turn backward 0.25 revolutions
    cowboy.RLeg.Rshin turn forward 0.25 revolutions
```

D.1.4. Methode *cowboy.nod()*

```
No variables
cowboy.Chest.neck turn forward 0.25 revolutions duration = 0.75 seconds
cowboy.Chest.neck turn backward 0.25 revolutions duration = 0.75
    seconds
```

D.2. Beschreibungstexte ohne Annotationen

In den folgenden Unterabschnitten sind die verschiedenen Beschreibungstexte zur Videosequenz „Cowboy“ angegeben.

D.2.1. Musterlösung

The ground is covered with sand. In the background is a pyramid. In the left front corner is a cowboy facing east. On the right side a bit further backwards there is a camel facing southwest. The camel nods twice and at the same time the cowboy walks 2 meters. Then the cowboy turns right by a quarter revolution. The cowboy jumps and claps his hands three times. The cowboy and the camel nod 4 times.

D.2.2. Beschreibungstext 1

The scene takes place in the desert. There is sand on the ground. In the background, there is a pyramid. There is a camel to the right. There is a cowboy to the left. The cowboy is facing west. The camel is facing the camera. The cowboy takes two steps forward. At the same time, the camel nods twice. The cowboy turns to the camera. He kicks and claps his hands three times. Both the cowboy and the camel nod four times.

D.2.3. Beschreibungstext 2

The ground is sandy. In the background is a pyramid with an little entrance in the middle down on the floor. In the foreground stands a cowboy on the left facing west. On the the ride side there is a camel in the middle facing the cowboy. The cowboy steps in the middle of the picture. His arms turns from up to down. The camel eats grass from the ground an then looks up to the cowboy. He now faces directly an his hands hang loose on the side of his coat. The cowboy jumps up an kick his left leg. Comes to the ground an jumps in the air again an again. Then the cowboy and the camel nod for four times.

D.2.4. Beschreibungstext 3

In the background of the desert there is a pyramid. On the left side in the foreground stands a cowboy. On the right side in the foreground stands a camel. The camel and the cowboy look at each other. The Cowboy takes three steps to the Camel. At the same time the camel nods twice. The cowboy turns right. Then he jumps, kicks and claps hands three times. Finally, the cowboy and the camel nod four times.

D.3. Cowboy-Beschreibungstexte mit annotierten Schleifen

Im Folgenden werden nun die Beschreibungstexte zum Video „Cowboy“inklusive ihrer *Loop*-Annotationen aufgeführt.

D.3.1. Musterlösung

```
1  The ground is covered with sand. In the background is a pyramid.
       In the left front corner is a cowboy facing east. On the right
        side a bit further backwards there is a camel facing
       southwest. The camel nods
2  <Loop LoopCycles="2" Verb="3">
3  twice
4  </Loop>
5  and at the same time the cowboy walks 2 meters. Then the cowboy
       turns right by a quarter revolution. The cowboy jumps and
       claps his hands
6  <Loop LoopCycles="3" Verb="3; 5">
7  three times
8  </Loop>
9  . The cowboy and the camel nod
10 <Loop LoopCycles="4" Verb="6">
11 4 times
12 </Loop>
13 .
```

D.3.2. Beschreibungstext 1

```
1  The scene takes place in the desert.
2  There is sand on the ground.
3  In the background, there is a pyramid.
4  There is a camel to the right.
5  There is a cowboy to the left.
6  The cowboy is facing west.
7  The camel is facing the camera.
8  The cowboy takes two steps forward.
9  At the same time, the camel nods
10 <Loop LoopCycles="2" Verb="8">
11 twice
12 </Loop>
13 .
14 The cowboy turns to the camera.
15 He kicks and claps his hands
16 <Loop LoopCycles="3" Verb="2; 4">
17 three times
18 </Loop>
19 .
20 Both the cowboy and the camel nod four times.
```

D.3.3. Beschreibungstext 2

```
1  The ground is sandy. In the background is a pyramid with an
       little entrance in the middle down on the floor. In the
       foreground stands a cowboy on the left facing west. On the the
```

```
    ride side there is a camel in the middle facing the cowboy.
   The cowboy steps in the middle of the picture. His arms turns
   from up to down. The camel eats grass from the ground an then
   looks up to the cowboy. He now faces directly an his hands
   hang loose on the side of his coat. The cowboy jumps up an
   kick his left leg. Comes to the ground an jumps in the air
   again an again. Then the cowboy and the camel nod for
<Loop LoopCycles="4">
four times
</Loop>
.
```

D.3.4. Beschreibungstext 3

```
In the background of the desert there is a pyramid. On the left
   side in the foreground stands a cowboy. On the right side in
   the foreground stands a camel. The camel and the cowboy look
   at each other. The Cowboy takes three steps to the Camel. At
   the same time the camel nods
<Loop LoopCycles="2" Verb="7">
twice
</Loop>
. The cowboy turns right. Then he jumps, kicks and claps hands
<Loop LoopCycles="3">
three times
</Loop>
. Finally, the cowboy and the camel nod
<Loop LoopCycles="4" Verb="8">
four times
</Loop>
.
```