Universität Trier
Computerlinguistik & Digital Humanities

Hausarbeit

zu dem Thema

Vorlage zur Erstellung einer Hausarbeit im Fach CL

1. Juli 2019

Vorname Name
Email: mein.name@uni-trier.de

Inhaltsverzeichnis

1 Einleitung

Chatbot Äugustaïst ein Chatbot, dessen Aufgabe es ist, seinem Nutzer Produkte aus dem Uni-Shop zu empfehlen. Dafür muss der Nutzer dem Bot Äugusta"via Texteingabe angeben, was gesucht wird. Diese Nutzereingaben werden mit Hilfe von Pattern-Matching analysiert und via PostgreSQL-Schnittstelle werden Antworten basierend auf Queries generiert, die dem geäuSSerten Wunsch des Nutzers entsprechen.

1.1 Ziele des Projekts

Vorrangiges Ziel des Projekts war der Versuch der Entwicklung eines computerlinguistischen Programms. Das Umsetzen im Studium angeeigneter Fähigkeiten theoretischer und praktischer Natur, wie z.B. Automatentheorie, Softwareentwurf und Datenbankentechnologie hat eine essentielle Rolle in der Konzeption und Realisierung des Programms gespielt. Weiteres Ziel war das Einarbeiten in ein fremdes Framework mit eigenem Interpreter, hier ChatScript (Wilcox 2019) von Bruce Wilcox, um ein entsprechendes Programm zu entwickeln.

1.2 Struktur der Arbeit

2 ChatScript: Grundlagen

Bei ChatScript handelt es sich um ein Framework, der es Entwicklern erlaubt, regelbasierte Chatbots zu entwickeln. Es handelt sich hierbei um "Natural Language tool"mit eigenem Interpreter, welcher Code in C++ übersezt.

2.1 ChatScript: Regeln

Regeln sind elementarer Bestandteil in ChatScript. In Regeln wird u.A. die Analyse von Nutzereingaben analysiert, Antworten des Bots festgelegt, Datenbankabfragen ausgeführt und der Gesprächsablauf gesteuert.

In Augusta finden sich Regeln in verschiedenen Formen wieder. Diese sind:

- t: für Ausgaben des Chatbots, ohne folgende Antwort
- u: für Ausgaben in Form von Fragen, das beduetet, dass auf Regeln mit u: eine Antwort des Nutzers folgen muss
- a:, b:, c: etc. für Antworten des Nutzers. Auf diese Regeln können Pattern-Matching-Ansätze angewandt werden, um Teile der Eingabe ggf. zu extrahieren.

Topics haben, auch wenn optional, meist einen Bezeichner, eine Bedingung zur Ausführung und es lassen sich dabei Variablen zuweisen. Ein abstraktes Beispiel:

Listing 2.1: Syntax für Regeln

```
1 t: BEZEICHNER (KONDITION) \$meineVariable=Wert Ausgabe des Bots
```

Im folgenden einige Beispiele für Regeln:

Listing 2.2: Beispiel für t:

```
1 t:  ( %input<%userfirstline )
2     ^keep()
3     [Hallo] [Hi] [Guten Tag], ich bin Augusta und kann dich beraten, wenn du e
```

In dieser Regel wird der Nutzer begrüSSt und im folgenden der Gesprächsverlauf zur Regel 'GREET' weitergeleitet.

Listing 2.3: Beispiel für Gesprächsablauf mit Antworten u:, a: und b:

```
1  u: INTRO (\$enter211) [Das ist gut] [Das ist toll] [Das freut mich], [dabei
2
3                  a: ( ~positiv ) \$introyes = 1a
4         # DATENBANK
```

```
5            if (^dbinit(dbname = postgres port = 5432 user = postgres passwor
6                {[Lass uns anfangen] [Super, auf geht's]!  ^reuse( FIRSTQ )
7            } else {dbinit failed - \$$db_error  ^reuse( FIRSTQ ) }
8        a: ( ~negativ ) Tut mir Leid, ich kann eigentlich nur beraten. Bist du
9            b: ( ~negativ )  \$introyes = 1a if (^dbinit(dbname = postgres
10                                   else {dbinit failed - \$$db_error ^reuse
11           b: ( ~positiv ) \$enterEnd2 = 1b Das ist schade. ^reuse( ~ende.ASK
```

In diesem Abschnitt fragt der Bot mit zufällig ausgewählter Frage, ob der Kunde etwas kaufen möchte. Falls die Eingabe des Kunden ein Wort aus dem Konzept positiv enthält, wird entsprechend die Datenbank geladen und in die Regel FIRSTQ im selben Topic aktiviert. Falls die Eingabe ein Wort aus dem Konzept negativ enthält, so wird gefragt, ob man nicht wirklich was sucht. Je nach Antwort darauf wird eine entsprechende Regel, d.h. eines der beiden b: Regeln ausgeführt.

2.2 ChatScript: Topics

Die Quelldateien in einem ChatScript-Programm werden auch als "Topics"bezeichnet und haben die Dateiendung .top. Topics können als Zusammenfassung mehrerer Regeln betrachtet werden. Es bietet sich an, Topics als Module eines Chatbots zu handhaben. So wird z.B. in diesem Projekt Äugusta"die Datenbankabfrage als eignenes Topic gehandhabt. Topics beginnen immer mit einer Tilde -äls Präfix gefolgt vom Topicnamen, meist als erste Zeile in einer .top-Datei oder vor Auflisting von Regeln:

Listing 2.4: Topicbezeichner in dbsearch.top

```
1 topic: ~dbsearch  [] (\$gosearch)
```

2.3 ChatScript: Konzepte

Syntaktisch werden Konzepte wie topics gehandhabt, das heiSSt, dass Konzepte mit Konzeptname referenziert werden. Ein Konzept kann als eine Menge von Synonymen verstanden werden, ähnlich zu Einträgen in einem Thesaurus. Obwohl es sich hier um Mengen handelt, stehen diese aus syntaktischer Sicht in Klammern () oder []. Ein Beispiel anhand des Konzepts "ciao":

Listing 2.5: Konzept 'ciao' aus konzepte.top

```
1 concept: ~ciao [ciao tschüss "good bye" bye "daje" "eddi un merci" "äddi a mei
```

Anzumerken ist, dass multiword expressions in Anführungszeichen stehen müssen, da einzelne Bestandteile dieser sonst als einzelne Elemente betrachtet werden.

Konzepte sind besonders wichtig in Äugusta", da es sich anbietet, das Pattern-Matching von Konzepten abhängig zu machen. So wird mit Hilfe von Konzepten das Arbeiten mit der PostgreSQL-Schnittstelle gewährleistet: Für jeden möglichen Eintrag pro Spalte in der PostgreSQL Datenbank existiert eine Konzept. Als Beispiel alle Einträge für die Spalte Änwendungszweck":

Listing 2.6: Konzept 'anwendungszweck' aus konzepte.top

```
1 concept: ~anwendungszweck [Unterhaltung Schreibwaren EssenTrinken Essentrinken
```

Der Nutzer beschreibt beispielsweise in einem Satz, welchen Anwendungszweck das gesuchte Produkt erfüllen soll. Wird ein Wort aus dem Konzept gefunden, so wird dieser als solches Erkannt, sofern vom Skript vorgesehen. Damit wird sichergestellt, dass in den Queries nur Eigenschaften vorkommen, die in Datenbankeinträgen existieren. Sollte der Anwender nach Eigenschaften suchen, die nicht existieren, so werden diese ignoriert (Siehe: Pattern-Matching, Query).

Des Weiteren erlaubt es ChatScript, mehrere Konzepte in einem Konzept zu vereinen. Ein solches Konzept ist als Vereinigung mehrerer Konzepte zu verstehen:

Listing 2.7: Konzept 'positivkaufen' aus konzepte.top

```
concept: ~positivkaufen [ ~yes ~zustimmung ~kaufen]
```

'Positivkaufen' enthält somit alle Wörter, die in yes, zustimmung und kaufen vorkommen.

2.4 ChatScript: Variablen

Variablen in Augusta kommen gröSSenteils in der Form $variablenname vor. $ vor dem Identifier bedeutet, dass diese Variable permanent ist und über Regeln hinaus gespeichert wird. Variablen werden für gewöhnlich im Kopf einer Regel instanziiert. Bei den zugewiesenen Werten kann es sich um Konstante handeln oder Werte die mit Hilfe von Pattern-Matching. Im folgenden Beispiel wird das erste Wort der Nutzereingabe, welches sich im Konzept geschenkidee befindet in der Variable $geschenkidee gespeichert.

Listing 2.8: Zuweisung des Wertes einer Variable

```
a: ( _~geschenkidee ) \$anwendungszweck = ^"'nichts'" \$geschenkidee = ^"'_0
```

Bereits instanziierte Variablen lassen sich einsehen mit dem Befehl $^{w}alkvariables(^{o}utputmacro)$.

Da das zurücksetzen von Variablen nicht trivial ist, lassen sich diese mit dem Befehl $^{r}eset(VARIABLES)zurcksetzen.D$

Listing 2.9: Zwischenspeichern einer Variable

```
        \$_cs_bottmp = \$cs_bot
    ^reset( VARIABLES )
    #....
        \$cs_bot = \$_cs_bottmp
```

Dieser Schritt findet in ende.top statt. Dies ist nötig, um zu gewährleisten, dass Informationen wie der Name des Kunden als auch die Insatz des Bots nicht verloren geht, wenn eine weitere Empfehlung im selben Gespräch erfolgen soll.

2.5 ChatScript: Variablen als Bedingungen zur Steuerung von Programmablauf

Nativ wählt ChatScript Regeln in einem Topic zufällig aus, um durch themenbezogene zufällige Antworten natürlich zu wirken. Da Augusta, ein Chatbot zur Kaufberatung, jedoch eine lineare Konversation gewährleisten muss, wird in fast allen Regeln mit Bedingungen gearbeitet. Ein Beispiel:

Listing 2.10: Regelkopf von FIRSTQ in kaufabsicht.top

```
1 u: FIRSTQ (\$introyes) Wir können nach etwas zum ...
```

Die Regel FIRSTQ darf nur aktiviert werden, sofern die Variable $introyes existiert.

Ähnlich lässt es sich auch verhindern, dass der Bot in eine Regel übergeht, indem man Variablen als Bedingungen stellt, die in keiner Regel instanziiert werden:

Listing 2.11: Regelkopf von STARTKAUF in kaufabsicht.top

```
1 t: STARTKAUF (\$enter999) ^reuse( INTRO)
```

Die Regel Startkauf ist gewöhnlich nicht betretbar für den Bot, da $enter999 als Variable nie instanziiert wird. Selbiges gilt für alle Regeln, dessen Bedingung die Existenz einer Variable mit '$enter...' erfordert. Diese Bedingungen dienen dazu, um ein zufälliges Springen des Bots zu verhindern. Stattdessen muss manuell von einer Regel auf diese weitergeleitet werden, was mit Hilfe von Befehlen zur Steuerung des Promgrammablaufs erfolgt.

Alternativ, jedoch nicht so häufig ist die Bedingung, dass ein Wort eines Konzepts zu erwähnen ist, damit eine Regel betreten wird:

Listing 2.12: Regelkopf einer Regel in kaufabsicht.top

```
1 b: (~willGeschenk) \$anwendungszweck = ^"'nichts'" Du möchtest etwas verschenk
```

Diese Regel, sofern erreichbar, erfordert von der Nutzereingabe, dass ein Wort aus dem Konzept 'willGeschenk' erwähnt wird. Ist dies nicht der Fall, so wird auch nicht in diese Regel übergegangen.

2.6 ChatScript: Befehle zur Steuerung von Programmablauf

In Augusta wurden zwei Methoden genutzt, um den Gesprächverlauf zu lenken. Dabei handelt es sich zum einen um $^{g}ambit(topic)undzumanderenum^{r}euse(topic.rule).ImLaufedesProjektshatsich^{r}euse(topic.rule)als$

Listing 2.13: Regel STARTKAUF in kaufabsicht.top

```
1 t: STARTKAUF (\$enter999) ^reuse( INTRO)
```

Wird die Regel 'STARTKAUF' betreten, so wird am Ende der Ausführung in die Regel 'INTRO' desselben Topics übergegangen.

Listing 2.14: Regel in keyexonesentence.top

```
1 a: (~positiv) \$gosearch = 1c Alles klar! Ich suche mal im Shop. d3 ^reuse( ~c
```

Sofern ein Wort aus dem Konzept ' positiv' aus der Nutzereingabe erkannt und ist diese Regel betretbar, so wird diese Regel ausgeführt und anschlieSSend die Regel 'WELCOME' in dbsearch.top, einem anderen Topic, betreten.

Im Verlauf der Entwicklung wurde $^{g}ambitmit^{r}euseersetzt,da^{r}euseeinePrzisereSteuerungdesProgrammablaufserlaub$

2.7 ChatScript: Pattern-Matching

Schreiben über:

ChatScripts Interpreter bietet einen Hauseigenen Pattern-Matcher zum Vergleichen von Nutzereingaben. Diese Ansätze eignen sich an erster Stelle dazu, Bedingungen für Regeln zu schreiben. Die einfachste Form, um eine Nutzereingabe abzugleichen, ist zu überprüfen, ob Nutzereingaben ein aus Konzepten bekanntes Token enthalten. Die folgende Regel überprüft, ob die Nutzereingabe ein Wort aus dem Konzept positiv enthält. Ist dies der Fall und ist die Regel erreichbar, so wird die folgende Regel ausgeführt:

Listing 2.15: Regel in kaufabsicht.top

```
1 c: ( ~positiv ) \$enterEnd2 = 1b Das ist schade. ^reuse( ~ende.ASKIFHAPPY )
```

Es besteht des Weiteren die Möglichkeit, eine Nutzereingabe auf mehrere Konzepte zu überprüfen:

Listing 2.16: Regel in kaufabsicht.top

```
1 b: ( [~no ~nicht] ) \$enterEnd2 = 1b In Ordnung. ^reuse( ~ende.ASKIFHAPPY )
```

AuSSerdem stellt ChatScript Mittel bereit, um Aussagen über die Reihenfolge von Wörtern in einer Nutzereingabe zu treffen. Im Folgenden wird überprüft, ob das Wort 'nicht' in derselben Nutzeringabe wie ein Wort aus dem Konzept sagen vorkommt. Dabei wird nicht eingeschränkt, an welchem Index die Wörter in der Nutzereingabe vorkommen, sodass Nutzereingaben wie 'Das will ich nicht verraten' oder 'Nicht das will ich verraten' von dieser Bedingung akzeptiert werden.

: Listing 2.17
Regel KEINE$_{V}ORSTELLUNGinintroductions.top$

```
1 a: KEINE_VORSTELLUNG (<<[sagen] nicht>>) [Das verstehe ich] [Das kann ich nach
```

Interessanter wird es, wenn Wörter aus Nutzereingaben extrahiert werden müssen. Für diesen Zweck bietet ChatScript Wildcards der Form *n. Diese erlauben es, Token an einer bestimmten Stelle zu extrahieren und ggf. in einer Variable zu speichern.

Listing 2.18: Regel in introductions.top

```
1 a: VORSTELLUNG ([heiSSe bin ist lautet] _*1 >)
2              if (\$cs_token == \$stdtoken)
3             {
4             \$cs_token = #DO_INTERJECTION_SPLITTING |
5                          #DO_SUBSTITUTE_SYSTEM | #DO_NUMBER_MERGE |
6                          #DO_PARSE
7             retry(SENTENCE)
8             }
9          \$kunde =  pos(noun '_0 proper)
```

Im Grunde wird hier das erste Wort nach 'heiSSe', 'bin', 'lautet' oder 'ist' genommen und nach Verarbeitung in der Variable kunde gespeichert.

Wildcards spielen in Augusta eine Zentrale Rolle, wenn es darum, geht, Kundenwünsche zu verarbeiten. In keyexonesentence.top finden sich verschiedene Muster zum Verständnis von Kundenwünschen. Da-

bei wird drauf geachtet, dass Wörter erwähnt werden, die in den Konzepten entsprechend vorkommen, sodass eine Query anhand der gegebenen Wörtern möglich ist:

Listing 2.19: Muster 1 in keyexonesentence.top

```
    a: ( !!~positiveinteger * _~thingsandart {in} _~ausfuehrung * {_~positive
)
    # a: ( < {Ich} {möchten wollen suchen} {ein eine einen} _~thingsandart *
_~ausfuehrung *  {für Preis Wert} * {_~positiveinteger} * )
        # Test, ob _0 Name oder Art ist
        if (pattern _0~things) {\$things = ^"''_0'" zusammenfassung = ^join(\$
        } else {\$art = ^"''_0'"  \$zusammenfassung = ^join(\$zusammenfassung
        \$ausfuehrung = ^"''_1'"
        # Optionaler Preis
        if ( _2 AND ^isnumber(_2) ) {\$preis = ^"''_2'" \$zusammenfassung = ^
        } else {\$zusammenfassung = ^join(\$zusammenfassung ^" Es soll in \$au
        1. MUSTER
        ^reuse( SUMMARY )
```

In diesem Pattern anfangs wird gesagt, dass eine positive Zahl nicht zu Beginn auftauchen darf. Es dürfen beliebig viele Token kommen, bis der erste Token aus dem Konzept thingsandart erwähnt wird, optionaler Weise gefolgt von einem 'in' und einem Begriff aus dem Konzept ausfuehrung. Darauf dürfen dann beliebig viele weitere Token folgen. Es ist freigestellt, ob darauf noch eine positive Zahl zur angabe von Preis kommt. Dabei stehen $_{0,1,2}$ *etc.frdaserste,zweite,dritteerkannteWortaufKonzepten.Imfolgendenwirdineiner Abfrageermittelt,obesichbeidemerstenWortumeinTokenausdemKonzept thingsoder arthandeltundentsprechende*

2.8 ChatScript: Zufällige Ausgabe

ChatScript erlaubt es, dass an einer Stelle mehrere Ausgaben durch den Bot möglich sind. Mögliche Ausgaben werden mit Hilfe von [] unterschieden:

u: INTRO ($enter211) [Das ist gut] [Das ist toll] [Das freut mich], [dabei kann ich dir helfen] [ich kann dir dabei helfen, etwas zu finden] [ich kann dich beraten]. Also dann, wollen wir loslegen DEBUG?

Listing 2.20: Regel in keyexonesentence.top

```
u: INTRO (\$enter211) [Das ist gut] [Das ist toll] [Das freut mich], [dabei ka
```

Hier wählt der Bot zufällig aus, ob 'Das ist gut', 'Das ist toll' etc. ausgegeben wird gefolgt von 'Also dann ...'.

3 Augusta: Programmlogik

Da ChatBot Augusta im Grunde ein Deterministischer Automat ist, bietet es sich an, den Programmablauf als Automaten zu beschreiben. Im Folgenden wird der Programmablauf als Automat auf .top-Dateien aufgeteilt dargestellt. Dabei beschreiben Zustände die jeweiligen Regeln in Augusta. Folgendes gilt:

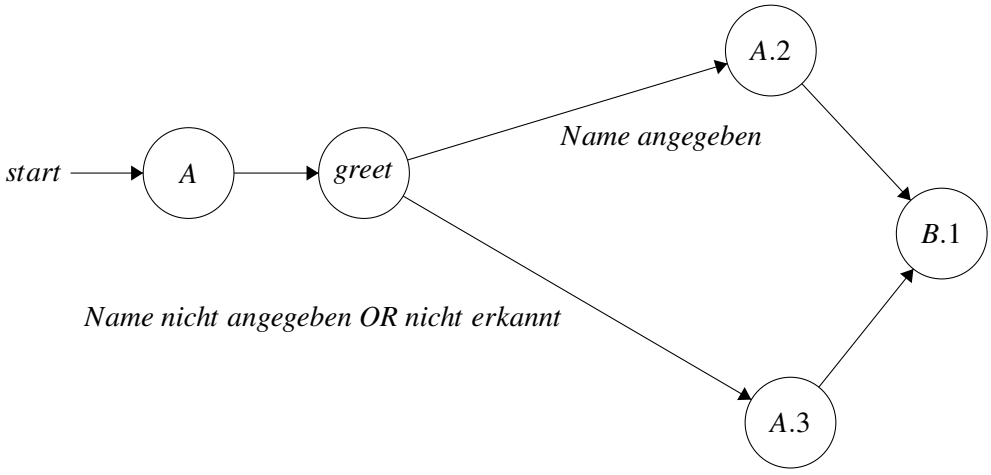
n wird in introductions.top

t introductions.vorstellung

s.KEINE$_V ORSTELLUNG$

1 ist kaufabsicht.Startkauf

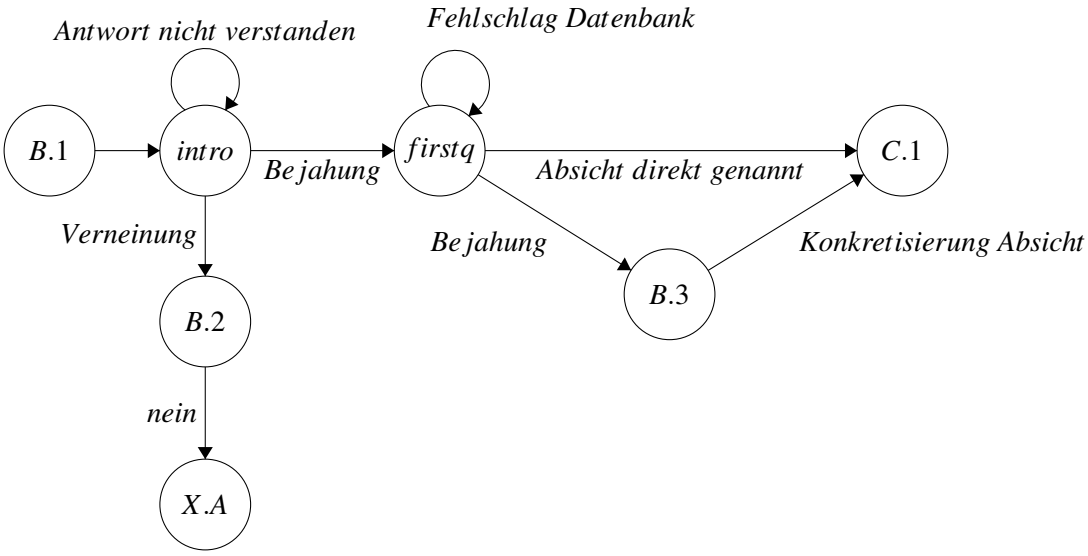
Für introductions gilt: Anfangs grüSSt der Bot den Nutzer, geht in Regel 'greet' über und fragt den Nutzer ob dieser seinen Namen verraten will. Je nachdem, ob aus der Nutzereingabe ein Name erkannt wird (siehe Abschnitt: Pattern-Matching) wird in die Regel 'vorstellung' bzw. 'keine$_v orstellung' bergegangen. Darauf hinwira$



Folgendes gilt:

1. B.1 ist startkauf
2. B.2 ist die Nachfrage, ob ein Kaufwunsch besteht
3. B.3 ist die Regel nach Angabe, dass man ein Geschenk bzw. etwas zu einer Anwendung will
4. X.A ist ENDE.ASKIFHAPPY
5. C.1 ist keyexprodukteigenschaften.startprodukteigenschaften bzw. keyexonesentence.startkauf
6. Ansonsten sind die Zustände nach ihren Regeln benannt

Von der Regel 'Startkauf' geht der Bot über in INTRO und fragt, ob ein Kaufwunsch besteht. Dies wird so lange erfragt, bis der Bot die Antwort des Nutzers verstanden hat. Ist dies nicht der Fall, erfolgt erneutige Nachfrage. Bestätigt der Nutzer, dass kein Kaufwunsch besteht, so wird dieser zu Regel 'ASKIFHAPPY' in Topic 'ENDE' weitergeleitet. Besteht doch ein Kaufwunsch, so wird in die Regel 'Firstq' weitergeleitet. Sollten Probleme mit der Initialisierung der Datenbank auftreten, so wird dieser Zustand so lange wiederholt, bis die Datenbank aktiviert werden kann. Daraufhin wird der Nutzer gefragt, ob dieser etwas zum Schenken oder etwas nach einem Zweck sucht. Nennt der Nutzer nur, dass er ein Geschenk bzw. etwas für einen Zweck sucht, so wird ihm mit Hilfe einer Datenbankquery vorgeschlagen, welche Geschnkzwecke bzw. Anwendungszwecke zur Verfügung stehen. Der Nutzer muss einen der aufgelisteten Geschenkideen/Anwendungszwecke nennen, welche als Variable abgespeichert werden. Nennt der Nutzer stattdessen direkt, was für eine Geschenkidee oder Anwendungszweck er möchte und ist dies auch bekannt, dann muss die Absicht nicht weiter konkretisiert werden und wird abgespeichert. Im folgenden betritt der Bot die Regel keyexprodukteigenschaften.startprodukteigenschaften bzw. keyexonesentence.startkauf, abhängig vom vorleigenden Ansatz (beide Ansätze im Projekt enthalten).

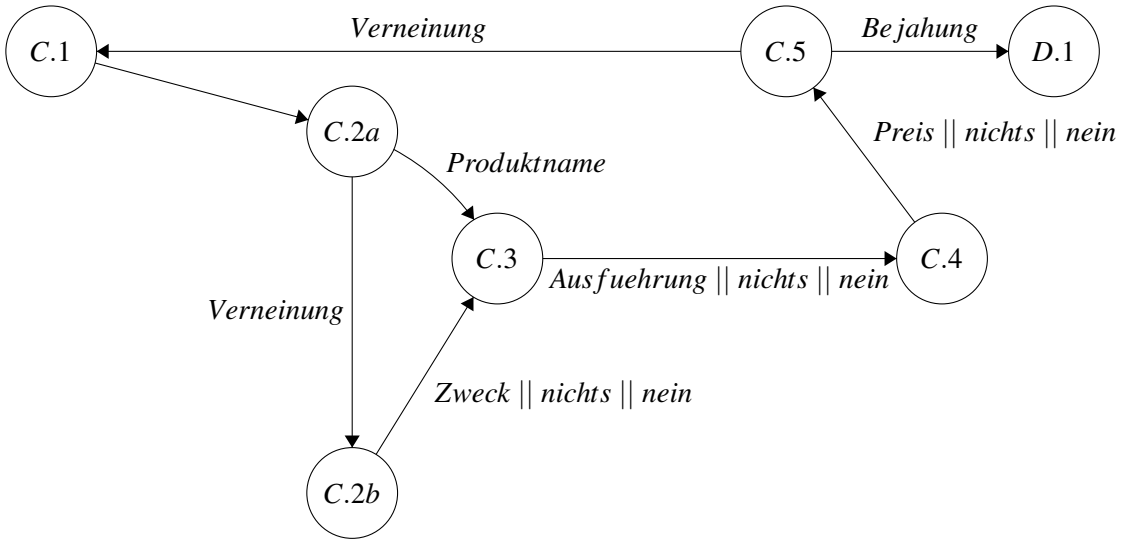


Folgendes gilt:

1. C.1 ist STARTPRODUKTEIGENSCHAFTEN
2. C.2a ist ASKNAME
3. C.2b ist KIND
4. C.3 ist DESIGN
5. C.4 ist PRICE
6. C.5 ist Summary
7. D.1 ist dbsearch.welcome

Wird der Ansatz Keyexprodukteigenschaften gewählt, so erfolgt ein Erfragen für die einzelnen Eigenschaften. In der Regel gilt: Werden in den Eingaben Stichworte erkannt, die als Eigenschaften in der Datenbank bekannt sind, so werden diese in Variablen gespeichert (siehe: Pattern-Matching) Zunächst wird gefragt, ob der Nutzer sein gewünschtes Produkt beim Produktnamen kennt. Ist dies nicht der Fall

oder wird die Eingabe nicht als solches identifiziert, wird gefragt, ob der Nutzer weiSS, von welcher 'Art', z.B. Kleidung, Verzehrmittel seine Idee ist. Erfolgt eine Eingabe, die im Konzept art bekannt ist, so wird in der entsprechenden Variable 'art' dieser Wert gespeichert. Daraufhin wird gefragt, ob man was zur Ausführung, z.B. Farbe oder Sprache sagen kann. Wird eine bekannte Ausführung erkannt, so wird diese gespeichert, ansonsten wird die Eingabe ignoriert. Im Anschluss wird gefragt, welchen Preis man maximal Zahlen will. Hierbei muss eine positive Integerzahl angegeben werden als Preislimit. In 'SUMMARY' wird zusammengefasst, was der Bot erkennen konnte und fragt den Nutzer, ob dies dem Nutzerwunsch entspricht. Ist dies der Fall, so betritt der Bot die Regel 'Welcome' in dbsearch, ansonsten fängt man bei 'startprodukteigenschaften' von Vorne an.

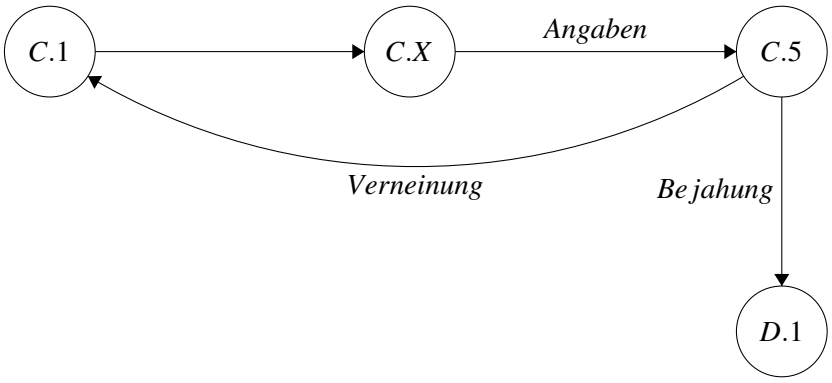


Keyexonesentence

Folgendes gilt:

1. C.1 ist STARTONESENTENCE
2. C.X ist quality
3. C.5 ist Summary
4. D.1 ist dbsearch welcome

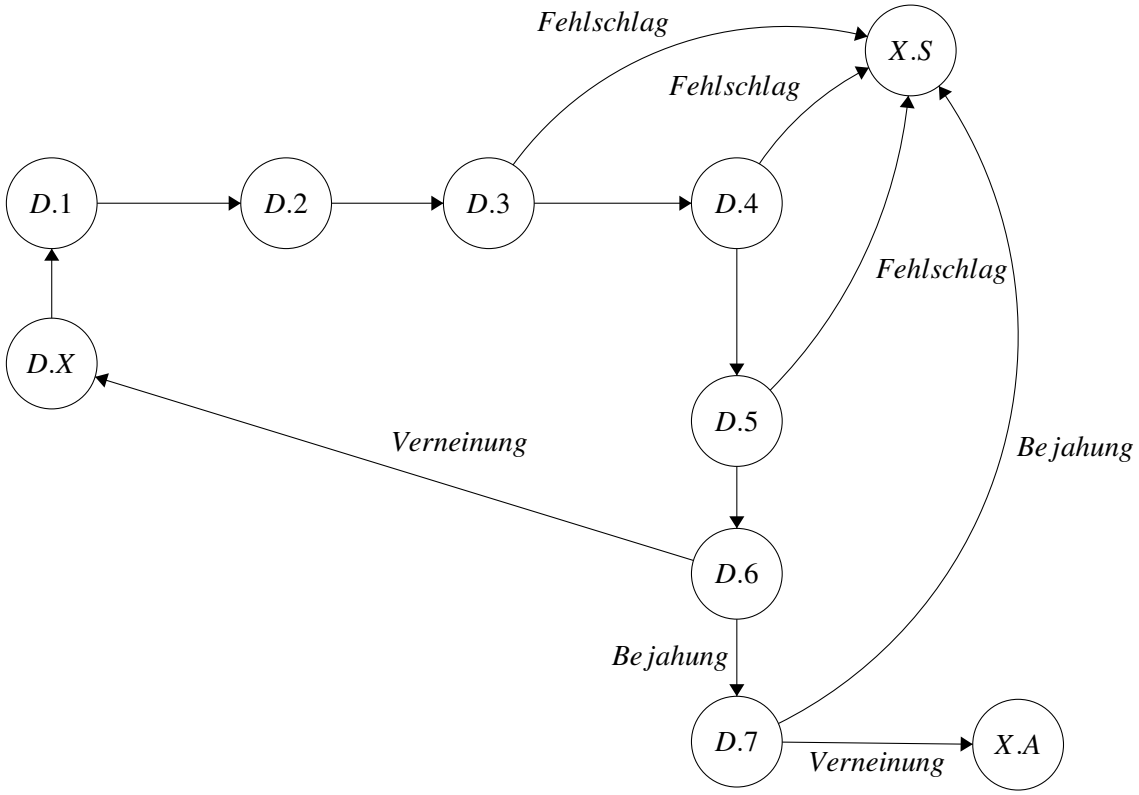
In Keyexonesentence kommt der im Abschnitt 'Pattern-Matching' besprochene Ansatz zur Extrahierung mehrer Informationen in einem Satz zum vorschein. Im Grunde wird in 'QUALITY' die Nutzereingabe genommen, ermittelt, um welches der 11 Muster es sich bei der Nutzereingabe handelt und die Informationen entsprechend extrahiert. In 'SUMMARY' wird zusammengefasst, was der Bot erkennen konnte und fragt den Nutzer, ob dies dem Nutzerwunsch entspricht. Ist dies der Fall, so betritt der Bot die Regel 'Welcome' in dbsearch, ansonsten fängt man bei 'startonesentence' von Vorne an.



Folgendes gilt:

1. D.1 ist WELCOME
2. D.2 ist SEARCHCREATE
3. D.3 ist SEARCHPRODUCT
4. D.4 ist PRODUCTNAME
5. D.5 ist PRODUCTDESCR
6. D.6 ist PRODUCTPRICE
7. D.7 ist RESPONSE
8. D.X ist SEARCHAGAIN
9. X.S ist ENDE.SETNULLREP
10. X.A ist ENDE.ASKIFHAPPY

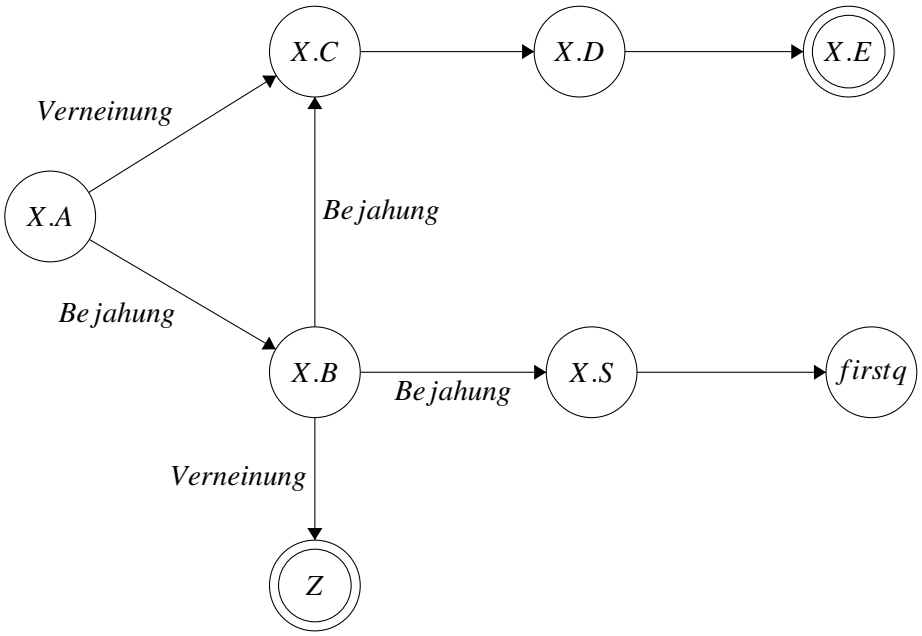
Basierend auf den ermittelten Nutzerinformationen wird nun eine Query erstellt. Dabei wird für jeden Nutzer eine Tabelle erstellt, die eine Kopie der originalen Datenbank ist. Diese wird im Schritt searchcreate erstellt. Daraufhin wird der Schritt Searchproduct betreten, in der alle Artikel, die der Beschreibung des Nutzers entsprechen. In den den folgeden drei Regeln ’PRODUCTNAME’, ’PRODUCTDESCR’ und ’PRODUCTPRICE’ jeweils Einträge aus Name, Beschreibung und Preis der Datenbank vorgestellt. In Response wird gefragt, ob dies dem Nutzerwunsch entspricht. Ist dies Nicht der Fall, so wird in ’SEARCHAGAIN’ übergegangen und erneut gesucht, wobei das bereits gefundene Produkt mit Hilfe der Eigenschaft ’queried’ in der Datenbank (siehe: Erläuterungen zur Datenbank) nicht erneut gefunden werden kann. Ist der Nutzer hingegen zufrieden, so wird dieser gefragt, ob dieser noch ein Produkt suchen will, ob dieser bereits gewähltes einsehen will oder ob dieser fertig ist. Ist ersteres der Fall, so wird in die Regel ’SETNULLANDREP’ im Topic ’Ende’ weitergeleitet, sodass eine erneute Suche möglich ist. Ist zweiteres der Fall, so wird in der Datenbank erneut anhand der ’queried’-Eigenschaft eine Anfrage für alle Produkte aufgerufen, die akzeptiert worden sind. Will der Nutzer kein weiteres Produkt mehr suchen, so wird die Regel ’ASKIFHAPPY’ in ’Ende’ betreten.



Folgendes gilt:

1. X.A ist askifhappy
2. X.B ist STILLHELP
3. X.C ist das Ablehnen einer weiteren Suche
4. X.D ist fgt
5. X.E ist goodbye
6. X.S ist SETNULLREP
7. firstq ist firstq aus Kaufabsicht
8. Z ist ein anderer Endzustand

Wird die Regel ’ASKIFHAPPY’ in ENDE betreten, so wird dem Nutzer die Frage gestellt, ob man noch was für den Kunden tun könne, oder nicht. Ist dies nicht der Fall, so wird im folgenden die Datenbank für den Nutzer gedroppt und der Bot Beendet (X.E). Ist dies jedoch der Fall, so wird in ’STILLHELP’ gefragt, ob man eine neue Suche starten wolle. Ist dies nicht der Fall, so wird der Bot beendet. Andererseits wird in ’SETNULLUNDREP’ übergegangen, wo für die Kaufberatung relevanten Variablen zurückgesetzt werden und anschlieSSend ’FIRSTQ’ im Topic ’Kaufabsicht’ betreten, zur erneuten Suche eines Produkts.


X.C
X.D
X.E
Verneinung
X.A
Bejahung
Bejahung
X.B
X.S
firstq
Bejahung
Verneinung
Z

4 Kapitel1j

4.1 UnterKapitel2j

4.2 UnterKapitel3j

5 Probleme

5.1 Zwei-Query Problem

Ein Problem, wofür ein Workaround gefunden werden konnte, war das 'Zwei-Query-Problem'. Dieses Problem tritt im Ansatz 'Onesentence' und allen davon abgeleiteten Ansätzen auf und tritt wegen einem der folgenden Fehler auf:

existierenden Datenbank

Fehlschlag in Queries

Ersteres tritt auf, wenn man mit Hilfe des Befehls ':build onesentence' o.Ä. den Bot, hier onesentence, neu kompiliert. Dies hat den Ursprung darin, dass der Bezeichner für die Datenbanken per Client mit einer Zufallszahl konkateniert wird und jeder 'Build'-Befehl dieselbe Zufallszahl generiert. Wird eine Tabelle nach Nutzung nicht gedroppt, dann tritt dieser Fehlschlag auf. Dieses Problem tritt in dieser Form nicht auSSerhalb des Testens auf, da es unwahrscheinlich ist, dass zwei Tabellen mit demselben Bezeichner 'zufallsgeneriert' werden, tritt dieses Problem nicht auf, wenn ein Bot unregelmäSSig, d.h. durch Verabschiedung in ende, geschlossen wird. Man kann dieses Problem manuell durch explizite drop-Befehle auSSerhalb von Chatscript behandeln. Findet eine solche Verabschiedung statt, tritt dies Problem nicht auf. Zweiteres trat auf, wenn mit den angegebenen Eigenschaften kein Produkt gefunden werden konnte. Wenn kein Produkt gefunden werden kann, ist laut Skript in den Zustand ' ende.SETNULLANDREP' überzugehen, um eine neue Suche zu gewährleisten. Nach der Programmlogik müssten hier die einzelnen Regeln bis zu der Datenbanksuche wieder iteriert werden, um Nutzerangaben zu sammeln. Stattdessen versuchte der Bot nach zurücksetzen der Variablen erneut eine Suche mit NULL-Variablen auszuführen. Dieses Problem wurde gröSSenteils umgangen, indem man den Nutzer mehrmals hintereinander fragt, ob man eine neue Suche ausführen will. Der Ursprung dieses Problems bleibt jedoch ungeklärt. Eine Vermutung ist, dass ChatScript intern Regeln, die bereits besucht worden sind, markiert, um nicht erneut von alleine in jene Regeln überzugehen Wilcox (2019).

5.2 Probleme mit der GUI

Bei dem Testen der GUI fallen einige Probleme auf:

1. Fehlender Support von PostgreSQL-Server für Mac und Linux
2. Server mit PostgreSQL-Unterstüzung vs. Server ohne PostgreSQL-Unterstüzung

Das Testen der GUI erfolgte durch das Hosten auf Localhost (127.0.0.1) der durch ChatScript bereitgestellten Dateien im Ordner 'SERVER BATCH FILES' (Wilcox 2019). Während das aufsetzen eines Servers auf Windows mit wenig Problemen verbunden war, fiel schnell auf, dass einige unvorhergesehene Probleme auftraten.

5.2.1 Fehlender Support von PostgreSQL-Server für Mac und Linux

In der Dokumentation für ChatScript wird aufgelistet, wie ein Server ohne Postgres-Unterstützung auf Windows, Mac und Linux-Distributionen aufzusetzen ist. Für Windows wird eine .bat Datei 'LocalPgServer.bat' beigeliefert wird, sodass das aufsetzen eines Servers auf Windows keine Probleme macht. Im Gegensatz dazu wird nichts über Server mit Postgres-Support für Linux oder Mac in der Dokumentation aufgelistet. Nach mehreren erfolglosen Versuchen wurde im Forum für chatbots nachgefragt, wobei diese Antwort unbeantwortet blieb.

5.2.2 Server mit PostgreSQL-Unterstüzung vs. Server ohne PostgreSQL-Unterstüzung

Ein weiteres Problem bei dem Test der GUI war, dass der Bot verschiedene Verhaltensweisen je nach .bat-Datei aufweist. So entspricht der Programmablauf des Bots dem der Konsolenausgabe, wenn ein Server ohne Postgres-Support, z.B. durch LocalServer.bat, eingerichtet wird. Hier verhält sich der Bot in den Zustandsübergängen identisch zu der Konsolenausgabe, mit dem Unterschied, dass der Nutzer die erste Eingabe machen muss bei Nutzung der GUI/des Webinterfaces. Das bedeutet, dass bis zum Zeitpunkt der Datenbankabfrage in dbsearch.top keine Probleme auftreten. Probleme treten in diesem Ansatz verständlicherweise erst dann auf, wenn eine Datenbankabfrage ausgeführt werden muss, da dieser Ansatz keine Unterstützung für Postgres liefert. Im Gegensatz dazu gab es in vieler Hinsicht Probleme mit der Einrichtung eines Servers mit Postgres-Unterstüzung. Bei einigen Anläufen startete der Bot gar nicht, im besten Fall wird dbsearch.response erreicht jedoch von dort aus erreicht der Bot keine weitere Regel, sondern iteriert weitergehend auf dbsearch.response. In vielen Versuchen erwiesen sich Übergänge in Regeln mit $^{r}euse()alsproblematisch.BeiAufsetzendesServersmitLocalPgServer.bat, istesvorgekomme$
$Untersttzungantwortet, konntenalleProblememitdemServermitPostgres-Untersttzungzeitnahnichtgelstwerden.Es$
$Untersttzungdazufhrt, dasseineweitereDatenbankberPostgresseitensdesServersohneEinflussvonEntwicklererstel$
$Postgres-Ablufesichbeidem ServermitPostgres-Untersttzungandersverhaltenalsohne Postgres-Untersttzung.$

6 Fazit

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

WILCOX, BRUCE (2019) *ChatScript*. https://github.com/ChatScript/ChatScript.