Universität Trier
Computerlinguistik & Digital Humanities

Dokumentation

zu dem Projekt

Chatbot Augusta

2. Juli 2019

Ali Kemal Yalama
E-Mail: s2alyala@uni-trier.de

Julia Roegner
E-Mail: s2juroeg@uni-trier.de

Inhaltsverzeichnis

1 Einleitung

„Augusta“ ist 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 von im Studium angeeigneter Fähigkeiten theoretischer und praktischer Natur, wie beispielsweise der Automatentheorie, Softwaretechnik 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 (**?**) von Bruce Wilcox, um ein entsprechendes Programm zu entwickeln.

1.2 Der Chatbot Augusta

Im Projektseminar der Computerlinguistik haben wir in kleinen Gruppen jeweils einen Chatbot für eine spezifische Anwendung entwickeln wollen. Unsere Gruppe, bestehend aus Ali Yalama und Julia Roegner, hat sich dazu entschieden, einen Chatbot als Kaufberatung für den Uni-Shop der Universität Trier zu bauen. Der Bot sollte für das Deutsche gelten und eine ausschlieSSlich beratende Funktion haben, d. h. Bestellungen oder Fragen zum Uni-Shop muss der Bot nicht beantworten können.
Als Basis diente der Beispiel-Dialog, der auch am Ende dieser Dokumentation abgedruckt ist. Hier wird davon ausgegangen, dass der Nutzer noch nicht weiSS, welches Produkt er haben möchte, sondern ihm nur bekannt ist, wofür er es möchte, also entweder eine Geschenkidee hat oder etwas für eine bestimmte Anwendung sucht. Dann bietet der Bot dem Nutzer die Kategorien an, die in der Datenbank für Anwendungsgebiet oder Geschenkidee hinterlegt sind. Im Weiteren kann der Kunde dem Chatbot noch weitere Informationen über den gesuchten Artikel wie eine Farbe oder den Preis übergeben, am besten in Form eines kompakten Satzes, den der Chatbot analysiert. Dann nutzt der Bot die extrahierten Stichwörter, um in einer Datenbank, in der die Informationen über die Produkte des Uni-Shops gespeichert sind, nach etwas Passenden zu suchen.
Der gefundene Gegenstand wird dem Kunden angeboten und der Kunde kann entscheiden, ob er ihn ablehnt, dann such der Bot erneut, oder annimmt, woraufhin der Bot fragt, ob eine neue Suche mit anderen Kriterien durchführen soll. Wenn nicht, beendet sich der Bot.
Der Name des Bots, Augusta, leitet sich vom Namen der antiken Stadt *Augusta Treveorum* an der Mosel her, aus der das heutige Trier hervorgegangen ist.

1.3 Struktur der Arbeit

2 Installation von Augusta

2.1 Installation

Zum Ausführen des Chatbots wird folgendes Benötigt:

ndows als Betriebssystem

stallation von PostgreSQL

nosten, z.B. auf Localhost

Obwohl ChatScript auch auf Mac und Linux laufen kann, existiert ein Problem bei ChatScript mit PostgreSQL-Support, dafür siehe Kapitel 'Probleme'. Sofern die GUI getestet werden will, muss dem Tester eine Möglichkeit bereit stehen, Webseiten, ggf. auf Localhost, zu hosten. Für Anmerkungen zu GUI siehe Kapitel 'Probleme'. Um den Bot zu testen, muss ChatScript runtergeladen werden auf `https://github.com/ChatScript/ChatScript`. Die Ordner 'FIRSTTRY', 'ONESENTENCE', 'ONESENTENCEGOODBYE' und 'ONESENTENCEGUIENDE' sind alle in den Ordner 'RAWDATA' zu kopieren. Dies gilt ebenso für die Dateien 'filesmine.txt', 'filesgoodbye.txt', 'filesonesentencegui.txt' und 'filesonesentenceguiende.txt', welche zum Kompilieren ihres zugewiesenen Ansatzes benötigt werden. Um ChatScript zu Starten, empfiehlt es sich, die Datei 'ChatScriptpg.exe' im Ordner 'BINARIES' auszuführen, für PostgreSQL-Support. Sollte der Wunsch bestehen, die GUI zu testen, so muss die .bat-Datei 'LocalPgServer' ausgeführt werden, um den Server zum Laufen zu bringen und die Dateien index.php und ui.php sollten in das jeweilige Verzeichnis für den Server kopiert werden. Für Anmerkungen, warum dies nur auf Windows erfolgt, siehe Kapitel 'Probleme'.

2.2 Versionen

In diesem Projekt finden sich verschiedene Ansätze für Chatbots wieder. Die Ansätze unterscheiden sich je nach Zweck verschieden stark.

2.2.1 FIRSTTRY

Der Ordner 'FIRSTTRY' enthält den ursprüunglichen Ansatz der Kaufempfehlung. Dieser Ansatz zeichnet sich dadurch aus, dass bei der Ermittlung des Kundenwunschs Satz für Satz erfragt wird, welche Eigenschaften das vom Kunden gewünschte Produkt haben soll. Dieser Ansatz lässt sich mit dem Befehl ':build mine' kompilieren. Für weitere Details zur Programmlogik und Ablauf dieses Ansatzes, siehe Kapitel: 'Augusta: Programmlogik'.

2.2.2 Onesentence

2.2.3 sec:Onesentence

Der Ansatz 'Onesentence' unterscheidet sich von FIRSTTRY in der Extraktion und Analyse des Kundenwunschs. Hier wird unter erweiterter Anwendung von Pattern-Matching-Verfahren, die von ChatScript aus bereitgestellt werden, der Kundenwunsch anhand eines Satzes extrahiert und analysiert. Dies soll einen intuitiveren Ansatz darstellen, da in natürlicher Sprache oft sogenannte Kundenwünsche in einem Satz beschrieben werden, ansatt als Antworten auf Konkrete fragen. Man kann diesen Ansatz mithilfe von ':build onesentence' kompilieren. Ansonsten sind Ansätze wie Vorstellung, Query und Interaktion nach Vorschlag an FIRSTTRY angelehnt.

2.2.4 OnesentenceGoodbye

Dieser Ansatz ist eine Modifizierung von 'Onesentence', die es dem Kunden erlaubt, zu jedem Zustand sich vom Bot zu verabschieden. Dies hat den Vorteil, dass ein 'drop'-Befehl die Tabelle, die dem Kunden zugeordnet wird, erfolgt und somit ein spätere Komplikationen auf Datenbankebene vermieden werden. Dieser Ansatz lässt sich mit ':build onesentencegoodbye' testen.

2.2.5 OnesentenceGui

Onesentencegui ist eine weitere Modifizierung von 'Onesentence'. Dieser Ansatz ist für die GUI gedacht, die erfordert, dass der Nutzer die erste Eingabe, wie z.B. 'Hallo' tätigt. Ansonsten unterscheidet sich die Programmlogik nicht von 'Onesentence'. Dieser Ansatz kann mit ':build onsentencegui' kompiliert werden, ist aber weniger für das Testen in der Konsole und am besten für die Benutzung in der GUI geeignet. Für Anmerkungen zu GUI, siehe Kapitel 'Probleme'. 'ONESENTENCEGUIENDE' ist der Ansatz 'OnesentenceGoodbye' für die GUI, d.h. es handelt sich hier um einen Ansatz für die GUI, die es dem Kunden Erlaubt, in jedem Zustand 'Tschüss' oder Ähnliches zu sagen, um den Bot zu beenden.

3 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.

3.1 Regeln

Regeln sind ein elementarer Bestandteil in ChatScript. In diesen Regeln wird unter Anderem 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 3.1: Syntax für Regeln

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

Im folgenden einige Beispiele für Regeln:

Listing 3.2: Beispiel für t:

```
1 t:  ( %input<%userfirstline )
2     ^keep()
3     [Hallo] [Hi] [Guten Tag], ich bin Augusta und kann dich beraten, wenn
          du etwas aus dem Online-Shop suchst. ^reuse( GREET )
```

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

Listing 3.3: Beispiel für Gesprächsablauf mit Antworten u:, a: und b:

```
1  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?
2
```

```
3                a: ( ~positiv ) \$introyes = 1a
4        # DATENBANK
5            if (^dbinit(dbname = postgres port = 5432 user = postgres
                 password = 1234))
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 dir sicher, dass du nichts aus dem Shop möchtest?
9                b: ( ~negativ )  \$introyes = 1a if (^dbinit(dbname =
                     postgres port = 5432 user = postgres password = 1234 ))
                      {Schön zu hören. ^reuse( FIRSTQ )}
10                                    else {dbinit failed - \$$db_error ^
                                         reuse(FIRSTQ)}
11           b: ( ~positiv ) \$enterEnd2 = 1b Das ist schade. ^reuse( ~ende
                 .ASKIFHAPPY )
```

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.

3.2 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 „Augusta“ die Datenbankabfrage als eignenes Topic gehandhabt. Topics beginnen immer mit einer Tilde („ “) als Präfix gefolgt vom Topicnamen, meist als erste Zeile in einer .top-Datei oder vor dem Auflisten von Regeln:

Listing 3.4: Topicbezeichner in dbsearch.top

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

3.3 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 3.5: Konzept 'ciao' aus konzepte.top

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

Anzumerken ist, dass Ausdrücke aus mehreren Wörtern, sogenannte „multiword expressions“, in Anführungszeichen stehen müssen, da die einzelnen Bestandteile dieser sonst als einzelne Elemente betrachtet werden.

In „Augusta“ sind Konzepte besonders wichtig, 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 ein Konzept. Als Beispiel alle Einträge für die Spalte „Anwendungszweck“ (zur Datenbank s. **??**):

Listing 3.6: Konzept 'anwendungszweck' aus konzepte.top

```
1 concept: ~anwendungszweck [Unterhaltung Schreibwaren EssenTrinken
     Essentrinken Alltag Accessoire]
```

Der Nutzer beschreibt beispielsweise in einem Satz, welchen Anwendungszweck das gesuchte Produkt erfüllen soll. Wird ein Wort aus dem Konzept gefunden, so wird dieses 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 (zum Pattern-Matching s. **??** und zur Query s. **??**).

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

Listing 3.7: Konzept 'positivkaufen' aus konzepte.top

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

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

3.4 ChatScript: Variablen

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

Listing 3.8: Zuweisung des Wertes einer Variable

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

Bereits instanziierte Variablen lassen sich einsehen mit dem Befehl `^walkvariables`(ôutputmacro) einsehen.

Da das Zurücksetzen von Variablen nicht trivial ist, lassen sich diese mit dem Befehl `^reset( VARIABLES )` zurücksetzen. Dieser Befehl setzt alle Variablen auf globaler Ebene zurück. Wenn man verhindern möchte, dass eine Variable zurückgesetzt wird, empfiehlt es sich diese als temporäre Variable zwischenzuspiechern, wie folgt:

Listing 3.9: Zwischenspeichern einer Variable

```
1         \$_cs_bottmp = \$cs_bot
2     ^reset( VARIABLES )
3     #....
4         \$cs_bot = \$_cs_bottmp
```

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

3.5 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 3.10: 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 3.11: 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 Programmablaufs erfolgt.

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

Listing 3.12: Regelkopf einer Regel in kaufabsicht.top

```
1 b: (~willGeschenk) \$anwendungszweck = ^"'nichts'" Du möchtest etwas
     verschenken. Ich kann dir folgende Kategorien anbieten:
```

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.

3.6 Befehle zur Steuerung von Programmablauf

In Augusta wurden zwei Methoden genutzt, um den Gesprächverlauf zu lenken. Dabei handelt es sich zum einen um `^gambit(~topic)` und zum anderen um `^reuse(~topic.rule)`. Im Laufe des

Projekts hat sich `^reuse(~topic.rule)` als praktischer herausgestellt, da man direkt in Regeln desselben Topics als auch in Regeln anderer Topics springen kann. Des Weiteren ignoriert `^reuse` die Bedingungen der Zielregel, sodass Regeln betreten werden, unabhängig davon, ob ein Wort des geforderten Konzepts oder die geforderte Variable existiert. Sofern sich eine Regel im selben Topic befindet, muss in `^reuse` nur die Zielregel angegeben werden:

Listing 3.13: 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 3.14: Regel in keyexonesentence.top

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

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 `^gambit` mit `^reuse` ersetzt, da `^reuse` eine präzisere Steuerung des Programmablaufs erlaubt. Während `^gambit` lediglich in eine anderes Topic springt und dieses von der ersten betretbaren Regel an ausführt, lässt sich durch `^reuse` festlegen, welche Regel zu betreten ist, ohne auf die Bedingung der Zielregel achten zu müssen.

3.7 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 3.15: 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 3.16: 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 3.17: Regel KEINE_VORSTELLUNG in introductions.top

```
a: KEINE_VORSTELLUNG (<<[sagen] nicht>>) [Das verstehe ich] [Das kann ich
   nachvollziehen]. ^reuse(~kaufabsicht.STARTKAUF)
```

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 3.18: Regel in introductions.top

```
a: VORSTELLUNG ([heiße bin ist lautet] _*1 >)
            if (\$cs_token == \$stdtoken)
           {
           \$cs_token = #DO_INTERJECTION_SPLITTING |
                       #DO_SUBSTITUTE_SYSTEM | #DO_NUMBER_MERGE |
                       #DO_PARSE
           retry(SENTENCE)
           }
        \$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. Dabei wird drauf geachtet, dass Wörter erwähnt werden, die in den Konzepten entsprechend vorkommen, sodass eine Query anhand der gegebenen Wörtern möglich ist:

Listing 3.19: Muster 1 in keyexonesentence.top

```
    a: ( !!~positiveinteger * _~thingsandart {in} _~ausfuehrung * {_~
       positiveinteger} *  )
    # a: ( < {Ich} {möchten wollen suchen} {ein eine einen} _~thingsandart
        * {in}  _~ausfuehrung *  {für Preis Wert} * {_~positiveinteger} *
       )
        # Test, ob _0 Name oder Art ist
        if (pattern _0~things) {\$things = ^"''_0'" zusammenfassung = ^
           join(\$zusammenfassung ^" Das Produkt heißt \$things.")
        } else {\$art = ^"''_0'"  \$zusammenfassung = ^join(\
           $zusammenfassung ^" Dein Produkt fällt unter den Obergriff \
           $art.")}
        \$ausfuehrung = ^"''_1'"
        # Optionaler Preis
        if ( _2 AND ^isnumber(_2) ) {\$preis = ^"''_2'" \$zusammenfassung
           = ^join(\$zusammenfassung ^" Es ist in \$ausfuehrung und
           maximal soll es \$preis Euro kosten.")
        } else {\$zusammenfassung = ^join(\$zusammenfassung ^" Es soll in
           \$ausfuehrung sein und eine preisliche Obergrenze hast du nicht
            angegeben.")}
        1. MUSTER
```

```
11          ^reuse( SUMMARY )
```

In diesem Pattern wird anfangs gesagt, dass eine positive Zahl nicht zu Beginn auftauchen darf. Es dürfen beliebig viele Token kommen, bis das 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. für das erste, zweite, dritte erkannte Wort auf Konzepten. Im Folgenden wird in einer if-Abfrage ermittelt, ob es sich bei dem ersten Wort um ein Token aus dem Konzept ' things' oder ' art' handelt und entsprechend eine Rückmeldung gegeben. Insgesamt 11 verschiedene Pattern dieser Art kommen in keyonesentence vor, aufgrund der Tatsache, dass der Nutzer frei angeben kann, wann er Angaben zu Name des Gegenstands Art, Ausführung oder Preis macht.

3.8 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 3.20: Regel in keyexonesentence.top

```
1 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?
```

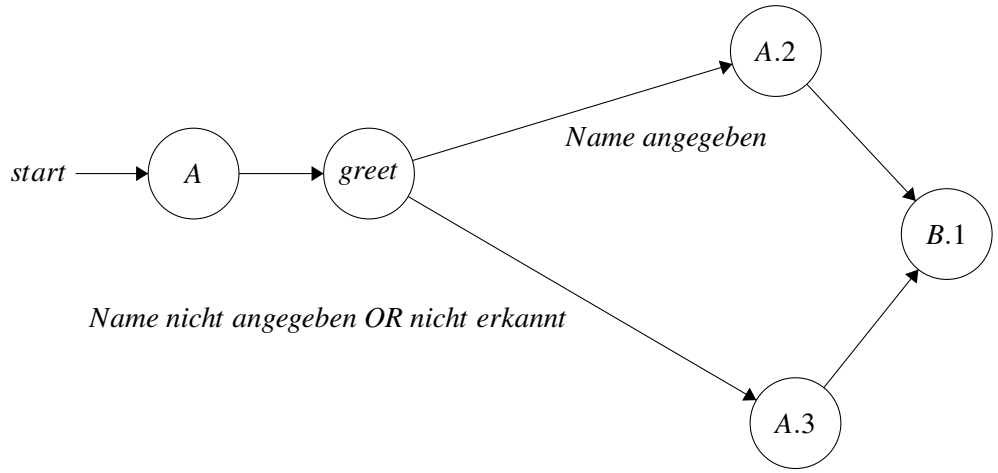
Hier wählt der Bot zufällig aus, ob 'Das ist gut', 'Das ist toll' etc. ausgegeben wird gefolgt von 'Also dann ...'.

4 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:

1. A ist die erste Regel die aufgerufen wird in introductions.top
2. A.2 ist introductions.vorstellung
3. A.3 ist introductions.KEINE_VORSTELLUNG
4. B.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_VORSTELLUNG' übergegangen. Daraufhin wird die Regel 'STARTKAUF' des Topics kaufabsicht betreten.

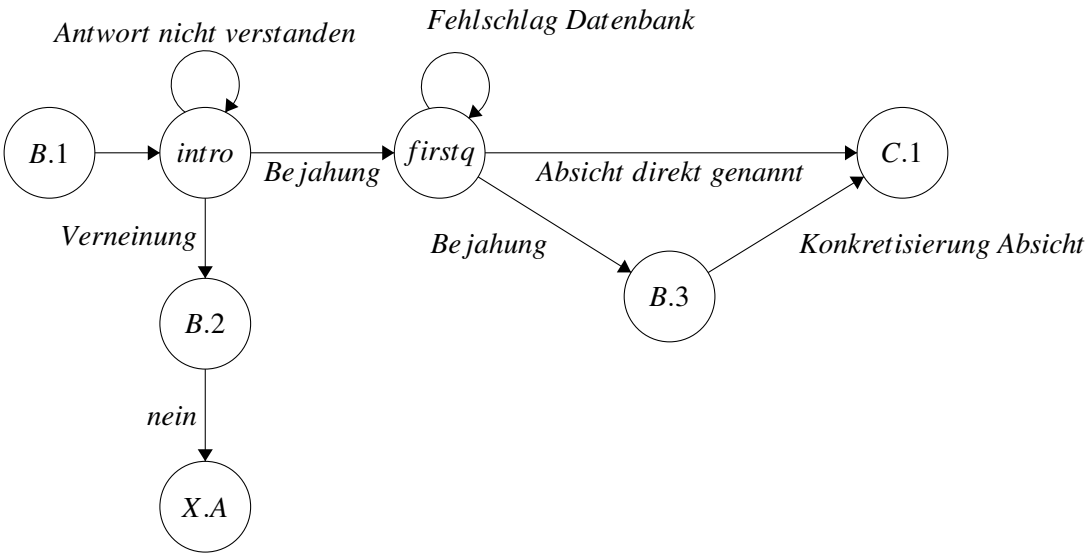


Folgendes gilt:

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 erneute 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 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 Geschenkzwecke 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 vorliegenden Ansatz (beide Ansätze im Projekt enthalten).

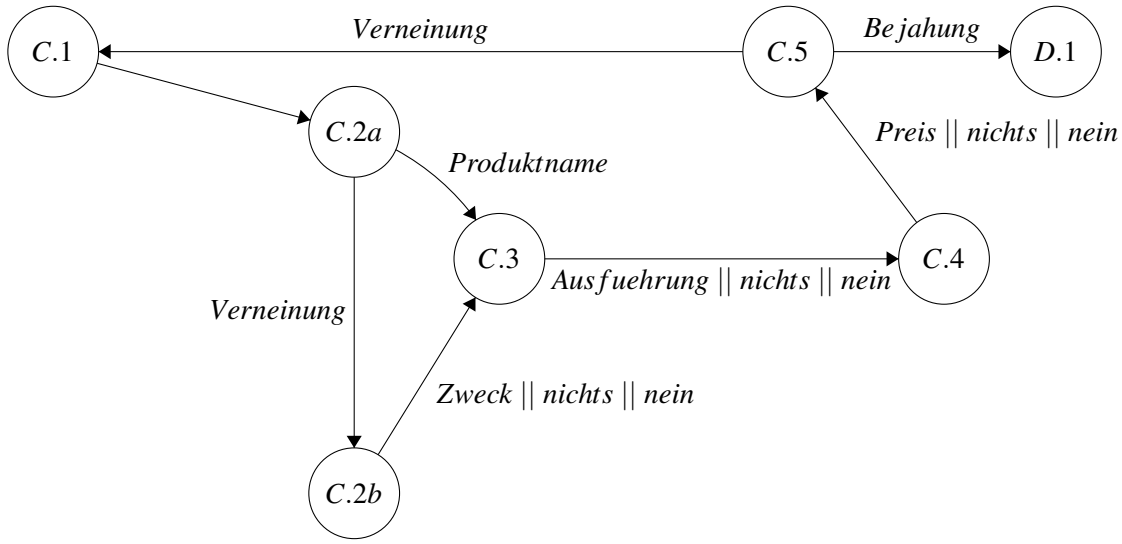


Folgendes gilt:

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 **??**) 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.

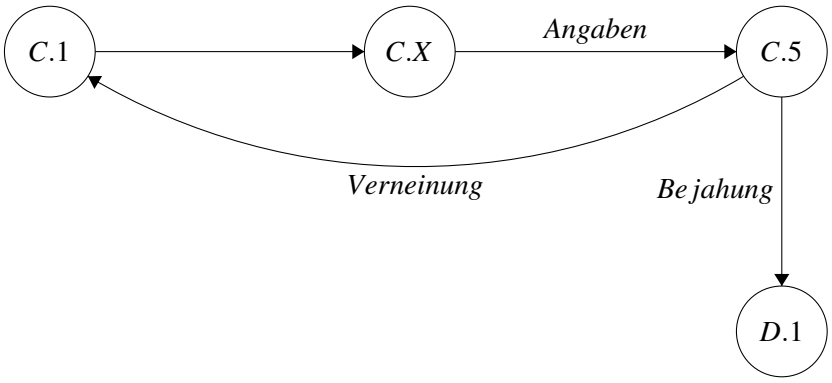


Keyexonesentence

Folgendes gilt:

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

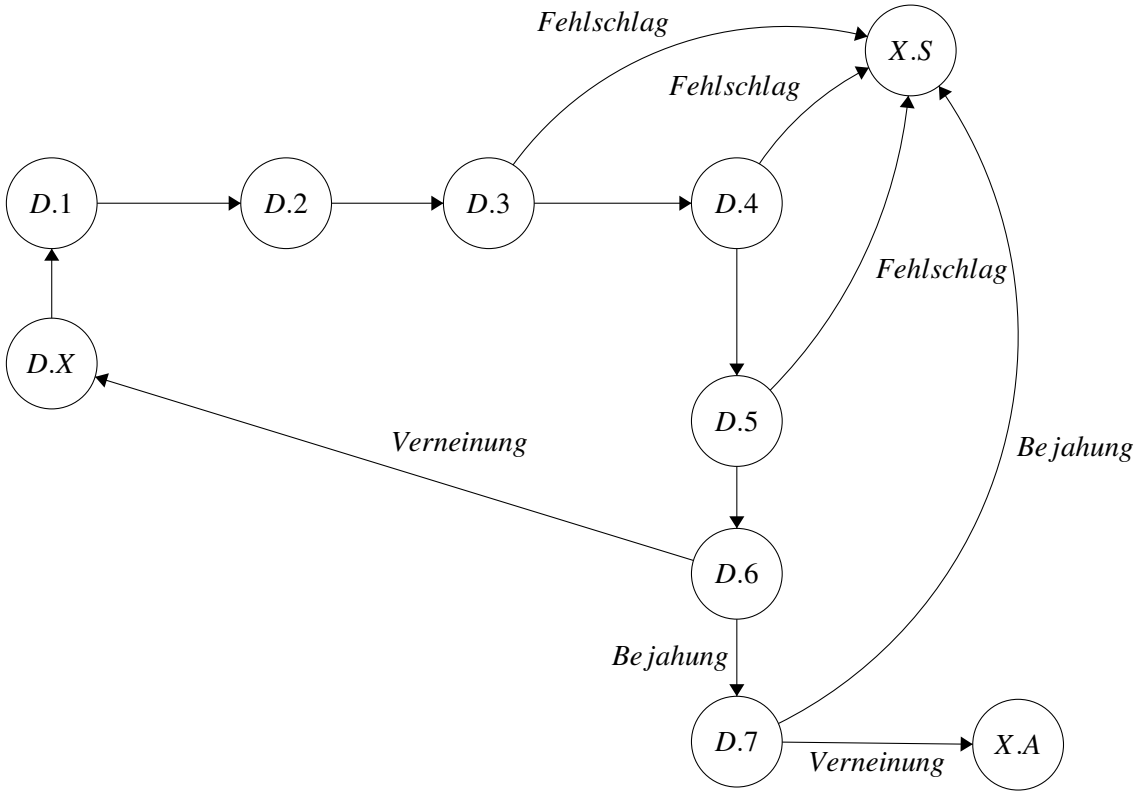
In Keyexonesentence kommt der im Abschnitt 'Pattern-Matching' besprochene Ansatz zur Extrahierung mehrerer Informationen in einem Satz zum Vorschein. Im Grunde wird in 'QUALITY' die Nutzereingabe genommen, ermittelt, um welches der 11 Muster es sich bei der Nutzereingabe handelt und die Informationen 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

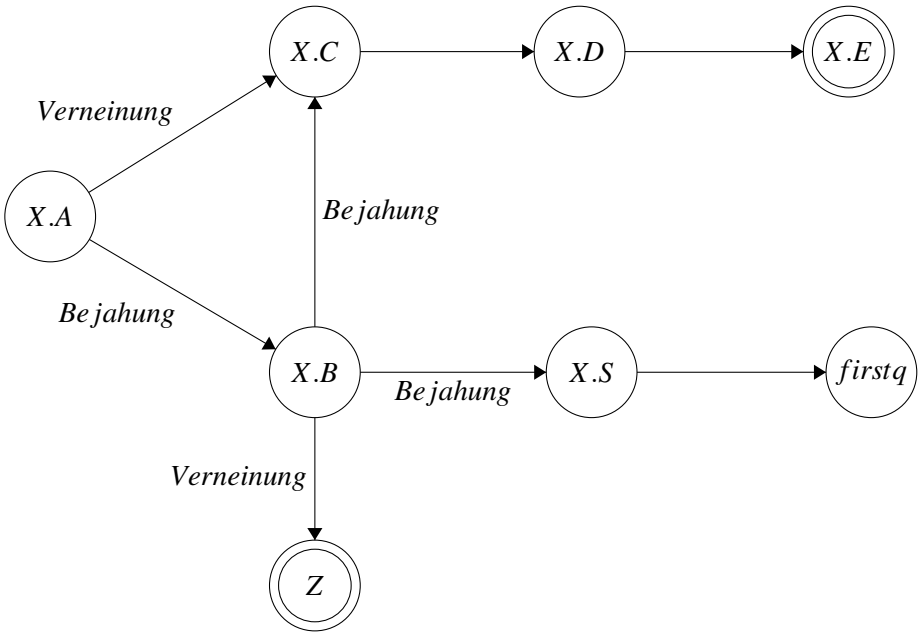
Basierend auf den ermittelten Nutzerinformationen wird nun eine Query erstellt. Dabei wird für jeden Nutzer eine Tabelle erstellt, die eine Kopie der originalen Datenbank ist. Diese wird im Schritt searchcreate erstellt. Daraufhin wird der Schritt SEARCHPRODUCT betreten, in der alle Artikel, die der Beschreibung des Nutzers entsprechen. In den den folgenden 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 **??**) 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

5 Datenbank und ihre Einbindung in den Chatbot

Herz des Dialogsystem ist eine Datenbank, in der die Produkte des Uni-Shops sowie Informationen zu ihnen wie Beschreibung oder Preis gespeichert ist. ChatScript unterstützt seit Version 4.2 PostgreSQL. Der folgende Abschnitt behandelt den Aufbau der Datenbank des Uni-Shops sowie ihre Einbindung in den Chatbot.

5.1 Aufbau der Datenbank

Für die Kundenberatung war eine Datenbank erforderlich, auf die der Bot zurückgreifen kann, um einzelne Produkte zu empfehlen. Die Datenbank Uni-Shop, die mittels des SQL-Skripts in *UniShop_Datenbank.sql* in PostgreSQL gebaut wird, besteht aus den drei Relationen *Artikel*, *Anwendung* und *Geschenk* sowie der Relation *PseudotabellePro*, die sich wiederum aus den drei anderen zusammensetzt. Während des Dialoges erstellt der ChatBot in Uni-Shop noch für jeden Nutzer eine weitere Relation, *mystringXXXX*, die vor der Beendigung des Bots automatisch gelöscht wird.
Die Daten der Artikel betreffend Name, Preis, Beschreibung und Ausführung sind gröSStenteils dem Uni-Shop der Universität Trier (https://www.uni-trier.de/index.php?id=50041, Stand: Mai 2019) entnommen. Alle Produkte, die dort angeboten werden, finden sich auch in der Datenbank. Um eine gröSSere Auswahl an Produkten zur Verfügung zu haben, was unteranderem dem Testen des Bots dienlich ist, wurden noch eigene, fiktive Einträge hinzugefügt. So gibt es den Einkaufschip, die Bücher und die Spiele nicht im originalen Universitätsshop. Zusätzlich wurde für einige Produkte, wie dem Schlüsselband oder der Kaffeetasse, noch eine weitere Farbe eingefügt.
Die Einträge für das Anwendungsgebiet und die Geschenkideen (s. **??** und **??**) beruhen auf unserer Vorstellung. Selbstverständlich sind hier andere Interpreationen möglich. Zudem kann die Datenbank jederzeit um weitere Einträge in den Relationen *Artikel*, *Anwendung* und *Geschenk* erweitert werden, wonach jedoch die Relation *PseudotabelePro* neu erstellt werden muss.
Da es bei den Relationen *Anwendung* und *Geschenk* mehrwertige Abhängigkeiten gibt, befindet sich die Datenbank in der vierten Normalform.

5.1.1 Die Relation *Artikel*

In der Relation *Artikel* sind die grundlegenden Informationen zu den einzelnen Produkten des Uni-Shops gespeichert. Jedes Produkt wird über eine eindeutige Artikelnummer, einen Namen, eine Kategorie (Art), einen Preis, eine kurze Beschreibung, sowie eine optionale Art der Ausführung näher bezeichnet.
Die Artikelnummer stellt, da sie für jeden Artikel einzigartig ist, den Schlüssel für ein Produkt dar. Die Nummer ist sechsstellig und setzt sich aus drei fortlaufenden Ziffern, einem Buchstaben für die Katego-

ArtNr	Name	Art	Preis	Beschreibung	Ausführung
001D01	Doktorhut	Deko	6.00	Schwarzer Doktorhut mit Siegel-Aufdruck der Universität Trier.	Schwarz
002A04	Schlüsselband	Alltag	2.00	Blau mit Logo und Internetadresse.	Blau
003A05	Schlüsselband	Alltag	2.00	Grün mit Logo und Internetadresse.	Grün
004D00	Wein-Set	Deko	9.50	Wein-Set mit Schachspiel „Checkmate“ bestehend aus Kellnermesser mit Gravur „Universität Trier“, Tropfring, Weinthermometer, Flaschenverschluss und Schachfiguren aus Holz, in schwarzer Holzbox mit Spielbrett auf dem Deckel.	

Tabelle 5.1: Auszug aus der Relation *Artikel*

rie und zwei Ziffern oder Buchstaben für die Ausführung zusammen.
Der Name der Produkte ist nicht eindeutig, es kommt also vor, dass Artikel in zum Beispiel verschiedenen Farben unterschiedliche Artikelnummern haben, aber denselben Namen tragen.
Die Kategorie des Produkt wird in der Relation mit *Art* bezeichnet. Hierbei handelt es sich um eine Art Oberbegriff, dem das Produkt zuzuordnen ist. So gehört die Kaffeetasse beispielsweise zur Kategorie „Alltag“ und das Notizbuch zum Oberbegriff „Uni“.
Der Preis, wird, wie im englischen Sprachraum üblich, mit einem Punkt, statt, wie in Deutschland, mit einem Komma angegeben, was dem der Struktur des PostgreSQl geschuldet ist. Zudem ist vorgegeben, dass der Preis nicht mehr als zwei Vorkomma-Stellen (d. h. kleiner als 99,99 Euro) sein darf.
Bei der Beschreibung ist es wichtig zu beachten, dass diese aus nicht mehr als 30 Wörtern insgesamt besteht, da sie andererseits nicht vollständig über den Chatbot ausgegeben werden kann. Zu näheren Informationen hierzu siehe **??**.
Die Ausführung behandelt bestimmte Eigenschaften eines Produkts, die für den Käufer relevant sind. Hierzu zählt die Farbe oder, bei Büchern, die Sprache. Zu beachten ist, dass nicht bei jedem Produkt ein Eintrag für die Ausführung vorliegt.
Die GröSSe eines Artikel, was im Fall der Kleidungsstücke für den Kunden von Bedeutung ist, wird in der Datenbank nicht berücksichtigt. Für diese bewusste Entscheidung gibt es mehrere Gründe: Da der Chatbot nur eine Kundenberatung darstellt und keine Bestellmöglichkeit beinhaltet, muss der Kunde persönlich beim Uni-Shop erscheinen, um das Produkt zu erwerben. Dort wird er über die vorhandenen GröSSen informiert und kann das Kleidungsstück möglicherweise auch anprobieren. Da Kleidung generell sehr unterschiedlich ausfallen kann, gerade bei Unisex-Kleidung, ist der Aussagewert der GröSSe in diesem Fall ohnehin als gering anzusehen. Desweiteren erteilt der ChatBot zu keinem Zeitpunkt eine Auskunft über die GröSSe.
Tabelle **??** stellt einen Auszug aus der Relation dar.

5.1.2 Die Relation *Anwendung*

In der Relation *Anwendung* sind Anwendungsvorschläge für die einzelnen Produkte notiert. Ebenso wie in der Relation *Geschenk* muss es nicht für jedes Produkt eine Anwendung geben und dasselbe Produkt kann mehreren Anwendungen zuzuordnen sein.

ArtNr	Anwendungsgebiet
001D01	Accessoire
002A04	Alltag
003A05	Alltag
004D00	EssenTrinken
004D00	Accessoire

Tabelle 5.2: Auszug aus der Relation *Anwendung*

ArtNr	Geschenkidee
001D01	Erinnerungsstück
002A04	Mitbringsel
003A05	Mitbringsel
004D00	Geschenk

Tabelle 5.3: Auszug aus der Relation *Geschenk*

Die Einträge in der Spalte ArtNr referenzieren die ArtNr der einzelnen Produkte in der Relation *Artikel*. Die Einträge in Anwendungsgebiet zeigen die Anwendungsmöglichkeiten. Derzeit gibt es fünf verschiedene Anwendungen: Accessoire, Alltag, EssenTrinken, Schreibwaren und Unterhaltung. Tabelle **??** zeigt einen Ausschnitt aus *Anwendung*.

5.1.3 Die Relation *Geschenk*

In der Relation *Geschenk* ist notiert, als was für ein Geschenk ein Artikel dienen könnte. Ebenso wie in der Relation *Anwendung* muss es nicht für jedes Produkt eine Geschenkidee geben und dasselbe Produkt kann mehreren Arten von Geschnken zuzuordnen sein.
Die Einträge in der Spalte ArtNr referenzieren die ArtNr der einzelnen Produkte in der Relation *Artikel*. Die Einträge in Geschenkidee zeigt, wofür das Produkt dienen kann. Derzeit gibt es vier verschiedene Geschenkarten: Erinnerungsstück, Gastgeschenk, Geschenk und Mitbringsel.
Im Falle des Geschenk ist zu beachten, dass dem Begriff hier eine Ambiguität zukommt: zum einen als Oberbegriff und zum anderen als Geschenkart im Sinne eines Präsents zum Geburtstag oder zu Weihnachten. Um desweiteren das Mitbringsel vom Geschenk unterscheiden zu können, wurde mit der internen Definition gearbeitet, dass ein Mitbringsel weniger als 6,00 Euro kostet.
Tabelle **??** zeigt einen Ausschnitt aus *Geschenk*.

5.1.4 Die Relation *PseudotabellePro*

Die Relation *PseudotabellePro* stellt die Grundlage für die späteren Relationen, die während des Dialogs speziell für den Nutzer erstellt werden.
Sie resultiert aus einem Natural Join von *Artikel* mit *Anwendung* und *Geschenk*, weshalb sie im Zuge der Aktualität nach einer Änderung in einer oder mehreren der drei Relationen neu erstellt werden muss.

AuSSerdem wird die Eigenschaft queried mit einer Integer als Wert (Default: 0) hinzugefügt. werden die Einträge, die Null sind, durch den String „nichts“ ersetzt. queried und das Ersetzen der Null-Werte ist für die Produktsuche von Bedeutung (s. **??**).
Da *PseudotanellePro* nur 60 Einträge umfasst, ist es in diesem Fall am einfachsten eine eben solche Relationen aus den anderen drei Relationen zu erstellen anstatt erst während der Suche Joins zum Beispiel nur *Artikel* und *Anwendung* auszuführen.

5.1.5 Die Relation *mystringXXXX*

Während des Dialoges und der Suche wird für jeden Nutzer eine eigene Relation erstellt, die *mystringXXXX* heiSSt. Der Name setzt sich aus dem String „mystring“ und einer zufällig gewählten, vierstelligen Zahl zusammen. Diese Zahl wird in Dokumentation als „XXXX“ wiedergegeben, daher die Betitlung *mystringXXXX*, während die Relation in der Datenbank beispielweise *mystring4083* heiSSen kann.
Eine eigene Relation für jeden Benutzer ist notwendig, da während der Produktsuche in der Relation die Einträge von queried individuell für jeden Nutzer verändert werden. Bevor sich der Chatbot in den Endzustand übergeht, wird *mystringXXXX* gelöscht.
Bei *mystringXXXX* handelt es sich um eine Kopie der Relation *PseudotabellePro*. Da die Datenbank Uni-Shop aufgrund der mehrwertigen Abhängigkeiten von Anwendungsgebiet und Geschenkidee in der vierten Normalform ist und *PseudotabellePro* alle drei grundlegenden Relationen in sich vereint, ist ArtNr als Schlüssel für die gesamte Relation nicht mehr ausreichend; der Schlüssel setzt sich nun aus ArtNr, Anwendungsgebiet und Geschenkidee zusammen. Da ein Schlüssel aus drei Eigenschaften hier die Suche unnötig verkomplizieren würde, wird noch die Eigenschaft row_number in *mystringXXXX* eingefügt, die jedes Objekt in der Datenbank eindeutig referenziert.

5.2 Einbindung von PostgreSQL in ChatScript

Seit Version 4.2 erlaubt ChatScript das einbinden von PostgreSQL-Datenbanken in den Chatbot. Dafür stellt ChatScript die Funktionen `^dbinit`, `^dbclose` und `^dbexecute` zur Verfügung. Im Folgenden werden die Funktionen kurz vorgestellt. Für weitere Informationen sei auf ChatScript PostgreSQL verwiesen. Dort findet sich auch eine kurze Einführung in PostgreSQL.

5.2.1 Die Funktionen `^dbinit` und `^dbclose`

Mit `^dbinit` wird eine Verbindung zu einer bereits existierenden Datenbank aufgebaut. Ohne diese Funktion schlagen alle weiteren Datenbank-Operationen fehl. Wenn bereits eine Verbindung zur Datenbank besteht und `^dbinit` erneut aufgerufen wird, schlägt dies ebenfalls fehl. Als ein Beispiel für `^dbinit` sei hier einer der Aufrufe aus *kaufabsicht.top* wiedergegeben:

Listing 5.1: Beispiel für `^dbinit`

```
1 if (^dbinit(dbname = Uni-Shop port = 5432 user = postgres password =
      1234))
2                 {[Lass uns anfangen] [Super, auf geht's]! ^reuse( FIRSTQ )
3             } else {dbinit failed - $$db_error ^reuse( FIRSTQ ) }
```

Die Datenbank, die mit `^dbinit` initialisiert wurde, bleibt die Datenbank, mit der gearbeitet wird, bis die Verbindung mit `^dbclose` geschlossen wird.

5.2.2 Die Funktion `^dbexecute` und Outputmacros

Um die SQL-Anweisungen aus ChatScript in PostgreSQL ausführen zu können, wird `^dbexecute` benötigt. Ein Aufruf könnte beispielsweise so aussehen:

Listing 5.2: Beispiel für `^dbexecute`

```
1 if (^dbexecute(^"SELECT DISTINCT anwendungsgebiet FROM anwendung;" '^
     dbFirst )) {}
2 else {dbexecute failed - $$db_error ^reuse( FIRSTQ ) }
```

Dieses Beispiel entstammt ebenfalls *kaufabsicht.top*. Der String (SELECT DISTINCT anwendungsgebiet FROM anwendung;) enthält dabei die Queries, während sich d̂bFirst auf das Outputmacro bezieht. Es ist dabei darauf zu achten, dass die einzelne Query mit einem Semikolon beendet wird und der Anweisungsblock insgesamt von doppelten Anführungszeichen wie im Beispiel vorgegeben eingerahmt wird. Durch die Verwendung der doppelten Anführungszeichen können die Werte von Variablen aus der ChatScript-Umgebung an SQL übergeben werden. Allerdings müssen die Variablen instanziiert sein, da eine Null-Variable in ChatScript in SQL als leerer String übersetzt wird. Zur Lösung dieses Problems in Augusta siehe **??**.
Das Outputmacro gibt vor, wie die Ergebnisse der SQL-Query ausgegeben werden. Im Prinzip handelt es sich dabei um eine selbstdefinierte Funktion am Kopf der .top-Dateien, bei der die einzelnen Wörter des SQL-Ergebnisses die Argumente darstellen. Die Zahl der Argumente ist hierbei auf maximal 31 beschränkt. Das folgende Beispiel für das Outputmacro ist recht einfach:

Listing 5.3: Beispiel für ein Outputmacro

```
1 outputmacro: ^dbFirst($_arg1)
2              $_arg1.
3              ^flushoutput()
```

`^dbFirst` ist für Suchergebnisse geschrieben, die pro Objekt nur aus einem Wort bestehen. Sollte es ein zweites Wort geben, wird dieses ignoriert. Hier wird lediglich nach jedem Wort ein Punkt gesetzt. Es sind aber auch kompliziertere Outputmacros möglich, die beispielsweise if-Anweisungen oder die zufällige Ausgabe von Wörtern und Sätzen (s. **??**) beinhalten. Bei SQL-Anweisungen, die keine Ausgabe beinhalten, wie beim Einfügen von Daten in eine Relation, ist die Angabe eines Outputmacros nicht erforderlich.

5.3 Der Gerbauch der Datenbank Uni-Shop in Augusta

Nach dem erfolgreichen Herstellen der Verbindung zu Uni-Shop in INTRO aus dem Topic Kaufabsicht ist die Datenbank einsatzbereit. Vor dem ersten Durchsuchen der Datenbank nach einem passenden Produkt wird in SEARCHCREATE die Relation *mystringXXXX* erstellt. Da während der Suche die Werte für queried individuell verändert werden, ist eine eigene Relation für jeden Benutzer unerlässlich.

queried	Bedeutung
0	Default-Wert. Das Produkt wurde weder ausgewählt noch verworfen. Nur Produkte mit queried = 0 können noch gefunden werden.
1	Das Produkt wurde während der aktuellen Suche gefunden und dem Kunden vorgeschlagen. Es wurde noch nicht angenommen oder abgelehnt.
2	Das Produkt wurde bei einer Suche gefunden, dem Kunden vorgeschlagen und abgelehnt. Bei weiteren SQL-Queries wird es ignoriert
3	Das Produkt wurde bei der aktuellen Suche gefunden und vom Kunden angenommen.
4	Das Produkt wurde in einer vergangenen Suche angenommen und wird dem Kunden beim Warenkorb angegeben.
5	Das Produkt wurde in einer vergangenen Suche angenommen, wird dem Kunden aber nicht im Warenkorb angegeben. Bei weiteren SQL-Queries wird es ignoriert.

Tabelle 5.4: Bedeutung der Werte für queried

mystringXXXX wird bei einem sauberen Programmende in ASKIFHAPPY gelöscht.
Zentraler Punkt der Suche ist die Eigenschaft queried. Sie funktioniert als Gedächtnis des Chatbots und stellt die Grundlage für die Ausgabe der Produktinformationen an den Benutzer und den Warenkorb dar. In der vorliegenden Version, wird die Datenbank erst zur Produktsuche im Topic Dbsearch verwendet, es ist allerdings auch möglich, die Anwendungsgebiete und Geschenkideen in FIRSTQ automatisch aus der Datenbank heraussuchen zu lassen. Näheres dazu siehe in **??**.

5.3.1 Die Eigenschaft queried

Obwohl ChatScript über die Möglichkeit verfügt, den geführten Dialog für jeden Nutzer einzeln zu speichern, erschien es uns einfacher und sinnvoller, die Informationen, welche Produkte schon gefunden, vom Kunden akzeptiert oder abgelehnt wurden, in der Datenbank selbst zu speichern. Dies geschieht in der Eigenschaft queried der Relation *mystringXXXX* (für *mystringXXXX* s. **??**).
queried ist vom Typ Integer und kann einen Wert von 0 bis einschlieSSlich 5 betragen. Zu Beginn, wenn noch keine Suche ausgeführt wurde, beträgt queried also bei jedem Objekt 0. Das Produkt, das den Suchkriterien entspricht und dem Kunden vorgeschlagen wird, wird mit 1 markiert. Da es dasselbe Produkt aufgrund der mehrwertigen Abhängigkeit mehrere Einträge in der Relation haben kann, ist es möglich, dass während eines Suchvorgangs mehrmals die 1 vorgeben wird. Mit einer 2 gekennzeichnete Artikel wurden vom Kunden abgelehnt. In Folge dessen werden diese auch in folgenden Suchen mit möglicherweise veränderten Kriterien dem Benutzer nicht mehr angezeigt.
Der Artikel, der der Kunde bei der aktuellen Suche ausgewählt hat, wird mit 3 kenntlich gemacht. In einem weiteren Schritt werden aus den Produkten mit 3 entweder eine 4 oder 5. Der Grund dafür liegt erneut darin, dass in *mystringXXXX* ein Produkt mehrere Einträge haben kann. Dem Kunden werden nur der Name, die Beschreibung und der Preis angezeigt. Da sich die Einträge aber in Anwendungsgebiet oder Geschenkidee unterscheiden, ist ein Unterschied für den Kunden nicht erkennbar und er wundert sich, weshalb ihm dieselbe Information mehrmals präsentiert wird oder hält es für einen Fehler. Um dies zu vermeiden, wird einer der mit 3 markierten Einträge mit 4 gekennzeichnet und die anderen mit 5. Nur der Eintrag des Produkts mit einer 4 wird dem Kunden bei Bedarf, zum Beispiel beim Waren-

korb gekennzeichnet.
Tabelle **??** bietet eine Übersicht der Bedeutung der queried-Werte. queried bildet die Grundlage der Suche. Es wird einmal nach einem Produkt gesucht und dieses daraufhin mit 1 gekennzeichnet. Alle darauffolgenden Operationen, die auf dieser Suche basieren, d. h. Ausgabe der Name und der Beschreibung, Markierung mit den anderen Zahlen und Ausgabe des Warenkorbs, erfolgt anschlieSSend auf der Basis des queried-Werts.

5.3.2 Die Query zur Produktsuche

Die eigentlich SQL-Query zur Suche hat noch keine Ausgabe, sondern verändert nur den Wert für queried auf 1. Die Query besteht aus zwei Teilen: Zuerst wird **ein** Artikel gesucht, dessen Eigenschaften den Suchkriterien entsprechen und in einem zweiten Schritt werden dann alle Einträge in *mystringXXXX*, die im Namen mit dem gefundenen Artikel übereinstimmen und daher das gleiche Produkt beschreiben, mit 1 markiert. Der folgende Programmcode zeigt die SQL-Abfrage:

```
1  UPDATE $tablename
2  SET queried = 1
3  WHERE name =
4    (SELECT name
5    FROM $tablename
6    WHERE
7      QUERIED = 0
8    AND
9      name = (CASE WHEN $things LIKE '%nichts%' THEN name ELSE $things END)
10   AND
11     art  = (CASE WHEN $art LIKE '%nichts%' THEN art ELSE $art END)
12   AND
13     preis <= (CASE WHEN $preis < 0 THEN preis ELSE $preis END)
14   AND
15     ausfuehrung = (CASE WHEN $ausfuehrung LIKE '%nichts%' THEN ausfuehrung
           ELSE $ausfuehrung END)
16   AND
17     geschenkidee = (CASE WHEN $geschenkidee LIKE '%nichts%' THEN
          geschenkidee ELSE $geschenkidee END)
18   AND
19     anwendungsgebiet = (CASE WHEN $anwendungszweck LIKE '%nichts%' THEN
          anwendungsgebiet ELSE $anwendungszweck END)
20   LIMIT 1);
```

In den Zeilen 4 bis 20 findet sich der erste Teil der Suche, in dem nach dem passenden Produkt gesucht wird, wobei nur Produkte berücksichtigt werden, die dem Kunden vorher noch nicht gezeigt worden sind (queried = 0).
Das Einbinden von Variablen aus der ChatScript-Umgebung in die SQL-Abfrage ist nicht unproblematisch, da Variablen mit NULL-Werten in SQl nicht als NULL sondern als leerer String interpretiert werden, was in der Folge zu einem Fehlschlagen der Abfrage führt. Um dennoch nicht für jeden der 68 möglichen Fälle eine eigene Abfrage schreiben zu müssen, haben wir eine Lösung gefunden, die dieses Problem umgeht.
Die Variablen für die extrahierten Stichwörter (`$anwendungszweck`, `$geschenkidee`, `$things`, `$art`

, `$preis` und `$ausfuehrung`) werden, wenn ihnen durch den Nutzer keine Kriterien zugewiesen werden, mit dem String „nichts“ bzw. bei `$preis` mit -1 initialisiert. In der Relation *mystringXXXX* wurden die Null-Werte ebenfalls durch den String „nichts“ ersetzt.
Um in PostgreSQL explizit anzugeben, dass eine Eigenschaft egal ist, kann dies so formuliert werden:

```
1 SELECT *
2 FROM table
3 WHERE property = property
```

Hier kommen für die Eigenschaft property also alle Werte in Frage, die es in der Relation für property gibt und damit ist `property = property` zu jedem Zeitpunkt wahr und bildet kein einschränkendes Kriterium zur Suche.
Die Abfrage in Augusta funktioniert ähnlich. Beim Betrachten von beispielsweise Z. 11 aus der SQL-Query oben fällt auf, dass das hier auch eine CASE-Anweisung enthalten ist.

```
1 art  = (CASE WHEN $art LIKE '%nichts%' THEN art ELSE $art END)
```

Es gibt also zwei Möglichkeiten, wie die Zeile die Suche beeinflussen kann: Wenn `$art` den String „nichts“ beinhaltet, was nur der Fall ist, wenn der Kunde keine Angaben zur Art des Produkts gemacht hat, lautet das Suchkriterium `art = art`. Hier würde die Produktauswahl nicht weiter eingeschränkt werden.
Ist nun aber `$art` ein anderer String als „nichts“ zugewiesen, dann lautet das Suchkriterium `art = $art`, sodass die Art des Artikels bei der Suche berücksichtigt wird. Analog werden die anderen Variablen gehandhabt.
Es wäre schön gewesen, wenn wir bei Name und Art ein LIKE hätten verwenden können, sodass bei der Eingabe von zum Beispiel „Tasse“ für Name die Kaffeetasse gefunden worden wäre. Leider ist das Verwenden von LIKE bei unserer Lösung nicht möglich, sodass die Werte in den Variablen exakt mit den Einträgen in der Tabelle übereinstimmen müssen.
Nach dem die Produkte gefunden wurden, wird das Ergebnis auf ein Produkt reduziert.
Der zweite Teil der Suche findet sich in Zeile 1 bis 3. queried wird in *mystringXXXX* dort auf 1 gesetzt, wo der Name des Eintrags mit dem des im ersten Teil gefundenen Produkts übereinstimmt. Wie bereits erwähnt, gibt es in der Relation aufgrund der mehrwertigen Abhängigkeit von Anwendungsgebiet und Geschenkidee für einige Produkte mehrere Einträge. Auch wird bei den Suchergebnissen nicht zwischen der Ausführung unterschieden, da dem Kunden ohnehin nur Name, Beschreibung und Preis angezeigt werden.

5.3.3 Die Ausgabe der Produkteigenschaften und der Warenkorb

In den Regeln PRODUCTNAME, PRODUCTDESCR und PRODUCTPRICE wird jeweils einzeln der Name, die Beschreibung bzw. der Preis des gefunden Produkts ausgegeben. Die SQL-Query für den Namen sei hier als Beispiel gegeben; die Abfragen für Beschreibung und Preis sind analog aufgebaut:

```
1 SELECT name FROM $tablename WHERE queried = 1 LIMIT 1;
```

Das Outputmacro `^dbsec` ist am Anfang von *dbsearch.top* definiert. Es sieht eine unterschiedliche Ausgabe für die Eigenschaften vor.
Zudem wird in `^dbsec` noch die Variable `$ergebnis` auf „positiv“ gesetzt. `$ergebnis` wird in PRODUCTNAME instanziiert und dient dazu, überprüfen zu können, ob überhaupt ein Schritt zuvor überhaupt ein Produkt gefunden wurde. Da in der Suche nur die Einträge in der Relation verändert werden,

aber standardmäSSig keine SQL-Ausgabe erfolgt, kann in ChatScript an dieser Stelle noch nicht festgestellt werden, ob ein Artikel gefunden wurde. Das passiert erst in PRODUCTNAME, wenn nach Einträge mit queried = 1 gesucht wird. Sind solche Einträge in der Relation vorhanden, wird entsprechend einer davon ausgewählt, `^dbsec` wird aufgerufen und `$ergebnis` wird „positiv“.
Wurde nun allerdings kein passender Artikel gefunden, so wird auf kein Eintrag mit queried = 1 gefunden. Dementsprechend wird weder das Outputmacro `^dbsec` aufgerufen noch `$ergebnis` verändert, woraufhin eine entsprechende Meldung an den Kunden weitergegeben und der Kunde zu FIRSTQ in dem Topic Kaufabsicht weitergeleitet wird.
Es gibt auSSerdem die Möglichkeit für den Kunden, dass er sich ansehen kann, welche Artikel er bereits akzeptiert hat. Die Möglichkeit in Anlehnung an die reale Welt und Internetshops als „Warenkorb“ bezeichnet und findet sich in der Regel REPONSE im Topic DBsearch.
Akzeptiert der Benutzer in RESPONSE das ihm angebotene Produkt, wird queried erst auf 3 und dort anschlieSSend bei einem Eintrag auf 4 und bei den anderen 5 gesetzt.
Beim Warenkorb werden zunächst die Artikel gezählt, die queried = 4 sind, um dem Kunden darüber informieren zu können, wieviele Produkte er bereits ausgewählt hat. In einem zweiten Schritt wird für diese Artikel dann Name und Preis ausgegeben. Da davon auszugehen ist, dass sich der Kunde noch erinnern kann, was er ausgewählt hat, wenn er den Namen des Produkts liest, ist die Ausgabe der Beschreibung hier nicht nötig. Die Ausgabe erfolgt über das Outputmacro `^dbThird`, das ebenfalls in *dbsearch.top* definiert ist.

5.3.4 Alternative Ausgabe von Anwendungsgebiet und Geschenkidee in FIRSTQ

In FIRSTQ werden die Werte, die für Anwendungsgebiet und Geschenkidee möglich sind, direkt in den Code geschrieben. Es gibt allerdings auch die Möglichkeit, diese automatisch aus der Datenbank extrahieren zu lassen und anschlieSSend dem Kunden zu präsentieren.
Da es nur 4 verschiedene Werte für Geschenkidee und 5 für Anwendungsgebiet gibt, haben wir uns der Einfachheit halber dagegen entschieden.
Wenn es in der Datenbank nun aber mehr Werte dafür gibt oder sich diese häufig ändern, kann es aus diesen oder anderen Gründen gewünscht sein, eine automatische Ausgabe durch den Chatbot zu generieren. Die Grundlage dafür ist FIRSTQ durchaus vorhanden.
Für die automatische Ausgabe müssen in der Regel FIRSTQ aus Kaufabsicht die Kommentarzeichen (#) vor den Datenbankabfragen

```
1 if (^dbexecute(^"SELECT DISTINCT geschenkidee FROM geschenk;" '^dbFirst ))
      {}
2 else {dbexecute failed list from table- $$db_error ^reuse( FIRSTQ ) }
```

und

```
1 if (^dbexecute(^"SELECT DISTINCT anwendungsgebiet FROM anwendung;" '^
    dbFirst )) {}
2 else {dbexecute failed list from table- $$db_error ^reuse( FIRSTQ ) }
```

sowie die Werte für Anwendungsgebiet und Geschenkidee direkt im Code entfernt werden. Dabei gilt es zu beachten, dass es die Datenbankabfragen jeweils zweimal in FIRSTQ gibt. AnschlieSSend müssen auch die Kommentarzeichen um das Outputmacro `^dbFirst` am Kopf der *kaufabsicht.top*-Datei gelöscht werden.
Damit die Änderungen übernommen werden, muss vor einem neuen Gebrauch der Bot neu kompiliert werden.

6 Probleme

6.1 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 **?**.

6.2 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' (**?**). Während das aufsetzen eines Servers auf Windows mit wenig Problemen verbunden war, fiel schnell auf, dass einige unvorhergesehene Probleme auftraten.

6.2.1 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.

6.2.2 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 `^reuse ()` als problematisch. Bei Aufsetzen des Servers mit LocalPgServer.bat, ist es vorgekommen, dass einige der vorgesehenen Regelübergänge per reuse nicht aktiviert wurden. Da Probleme mit LocalPgServer.bat nicht in der Dokumentation von ChatScript behandelt werden und der Macher und Entwickler von ChatScript, Bruce Wilcox, nicht auf Nachfragen mit dem Hosten eines Servers mit Postgres-Unterstützung antwortet, konnten alle Probleme mit dem Server mit Postgres-Unterstützung zeitnah nicht gelöst werden. Es liegt die Vermutung nahe, dass das Hosten eines Servers mit Postgres-Unterstützung dazu führt, dass eine weitere Datenbank über Postgres seitens des Servers ohne Einfluss von Entwickler erstellt wird, was zu allerlei Konflikten führt. Aufgrund Zeitmangels, mangelnder Dokumentation und fehlender Einsicht in die Binärdateien bzw. Konfiguration ist dieses Problem bis dato nicht gelöst. Die Fehlerquelle konnte nicht ausfindig gemacht werden und es ist ein Rätsel, warum auch nicht-Postgres-Abläufe sich bei dem Server mit Postgres-Unterstützung anders verhalten als ohne Postgres-Unterstützung.

7 Fazit

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

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