

# **A Chatbot Dialogue Manager**

Chatbots and Dialogue Systems: A Hybrid Approach

A.F. van Woudenberg, 850272823

June 17, 2014



**A Chatbot Dialogue Manager**

Chatbots and Dialogue Systems: A Hybrid Approach

A.F. van Woudenberg, 850272823

June 17, 2014

Master Education in Computer Science

T76318

Graduation committee:

Supervisor: Dr. E. Roubtsova

Examiner: Prof. Dr. S.M.M. Joosten

**Open University of the Netherlands**  
**Faculty of Management, Science and Technology**

# Acknowledgements

Firstly, I would like to thank my supervisor Dr. Ella Roubtsova for her advice, great enthusiasm and patience and for allowing me to pursue my own interests. I would also like to thank Prof. Dr. Stef Joosten for his feedback.

My thanks go out to Mathilde, Lennart and Fleur as well for participating in my experiments.

## **Abstract**

Conversational software, that is software with which a user can converse in a natural language such as English or Dutch, can be classified into two distinct categories: chatbots and dialogue systems. Chatbots work according to the principle of pattern matching where user input is matched to a fixed response.

In dialogue systems, user input is parsed into some semantical representation. This representation is then used by a component called a dialogue manager to determine what the response should be. A dialogue manager maintains a state of the conversation by tracking who spoke last, what knowledge is private and shared, the current goal, what plan should be followed to resolve an open issue and how much of the user input was understood.

Both categories of systems have their advantages and disadvantages. Chatbot systems are trivial to build and maintain, but are too simplistic for applications that do more than answering frequently asked questions. Dialogue systems on the other hand are harder to develop and maintain, can deal with less variety in user input, but are capable of handling and generating all kinds of linguistic phenomena such as grounding and information revision.

This thesis presents a system that combines ideas from chatbots and dialogue system. This hybrid system doesn't parse natural language into a semantic presentation, but contains a dialogue manager that can handle natural language directly. This system would make it easy to implement and maintain new domains even when the developer has little or no knowledge of computational linguistics. The system would also be able to deal with a variety of linguistic phenomena.

A statistics tutor has been implemented using this hybrid system with which three students interacted. This shows that it's reasonably easy to build and maintain new domains using this approach. It is also shown that the statistics tutor is able to deal with phenomena such as grounding, question accommodation and information revision.

## Samenvatting

Computersystemen waarmee in een natuurlijke taal, zoals het Engels of het Nederlands, gecommuniceerd kan worden, zijn in twee klassen in te delen. De ene klasse bestaat uit de zogenaamde chatbots, terwijl de andere klasse dialoogsystemen zijn. Chatbots werken op basis van patroonherkenning. Wanneer gebruikersinvoer voldoet aan een bepaald patroon dan zal de chatbot het antwoord dat bij dit patroon hoort als reactie geven.

Bij dialoogsystemen wordt gebruikersinvoer geanalyseerd en omgezet naar een semantische representatie. Deze semantische representatie wordt vervolgens gebruikt door een dialoogmanager om te bepalen wat de volgende (taal)handeling moet worden. Een dialoogmanager houdt een status bij van de dialoog. Deze status bevat gegevens zoals wie er voor het laatst heeft gesproken, welke kennis gemeenschappelijk is en welke niet, wat het huidige doel is, welke stappen er genomen moeten worden om een vraag te beantwoorden en tot in hoeverre gebruikersinvoer begrepen is.

Beide klassen van systemen hebben voor- en nadelen. Chatbots zijn eenvoudig te implementeren en onderhouden, maar zijn te simplistisch voor toepassingen die verder gaan dan het antwoorden op veelgestelde vragen. Dialoogsystemen daarentegen zijn lastiger te ontwikkelen en onderhouden, kunnen omgaan met een veel kleiner scala aan gebruikersinvoer, maar zijn wel in staat tot het genereren en herkennen van allerlei soorten taalverschijnselen zoals *grounding* en het herzien van eerder gegeven antwoorden.

Deze scriptie beschrijft een systeem dat het midden houdt tussen een chatbot en een dialoogstelsel. Gebruikersinvoer wordt hierbij niet omgezet in een semantische representatie. In plaats daarvan bevat dit hybride systeem een dialoogmanager die direct werkt met natuurlijke taal. Verwachte eigenschappen van dit systeem zijn dat het eenvoudig is om nieuwe domeinen te implementeren en te onderhouden zonder al teveel specialistische kennis. Ook zou dit nieuwe systeem om moeten kunnen gaan met allerlei taalverschijnselen.

Er is een statistiek tutor geïmplementeerd binnen dit hybride systeem. Drie studenten hebben vervolgens met dit systeem geconverseerd. Op basis hiervan kan onder meer worden geconcludeerd dat het betrekkelijk eenvoudig is een dergelijk systeem te bouwen en te onderhouden. Ook is aangetoond aan de hand van een mogelijke interactie dat het systeem in staat is tot het omgaan met taalverschijnselen zoals *grounding*, het herzien van eerder gegeven antwoorden, en *question accommodation*.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Chatbots and dialogue management . . . . .	1
1.1.1	ELIZA . . . . .	1
1.1.2	Research dialogue systems . . . . .	2
1.1.3	Information state approach . . . . .	2
1.2	Problem statement . . . . .	3
1.3	Aims of this project . . . . .	3
1.4	Research questions . . . . .	4
1.5	Overall approach . . . . .	4
1.6	Thesis outline . . . . .	5
<b>2</b>	<b>Chatbot technology and human deception</b>	<b>6</b>
2.1	The imitation game . . . . .	6
2.2	Some early systems . . . . .	7
2.3	Trickery and the Loebner contest . . . . .	8
2.4	ALICE . . . . .	10
2.5	Elements of AIML . . . . .	10
2.5.1	Categories . . . . .	10
2.5.2	Recursion . . . . .	11
2.5.3	Context . . . . .	12
2.5.4	Variables . . . . .	13
2.5.5	Pronoun swapping . . . . .	13
<b>3</b>	<b>Dialogue systems</b>	<b>15</b>
3.1	Speech acts . . . . .	15
3.2	Grounding . . . . .	16
3.3	Implicature . . . . .	17
3.4	Dialogue system components . . . . .	18
3.4.1	Speech recognition . . . . .	18
3.4.2	Text-to-speech synthesis . . . . .	19
3.4.3	Natural language understanding . . . . .	19
3.4.4	Natural language generation . . . . .	21
3.4.5	Dialogue manager . . . . .	21

3.4.6	Task manager . . . . .	25
3.5	The information state approach . . . . .	25
3.5.1	TrindiKit . . . . .	25
3.5.2	GoDiS . . . . .	27
<b>4</b>	<b>Methods</b>	<b>41</b>
4.1	Features and properties . . . . .	41
4.2	Case study . . . . .	42
4.3	Implementation . . . . .	43
4.4	Procedure . . . . .	44
4.5	Instructions . . . . .	44
4.6	Evaluation . . . . .	45
<b>5</b>	<b>Implementation</b>	<b>47</b>
5.1	SWI-Prolog . . . . .	47
5.2	Pattern matching . . . . .	48
5.3	Rule engine . . . . .	49
5.4	Information state . . . . .	50
5.5	System architecture . . . . .	52
5.5.1	Interpretation . . . . .	52
5.5.2	Generation . . . . .	57
5.5.3	Dialogue Manager . . . . .	59
5.6	Implementation of the case study . . . . .	68
5.6.1	Decision tree . . . . .	68
5.6.2	Plans . . . . .	68
5.6.3	Additional units . . . . .	70
5.6.4	Facts . . . . .	73
5.7	Implementation of the demo interaction . . . . .	73
<b>6</b>	<b>Testing of the system</b>	<b>74</b>
6.1	Participant #1 . . . . .	74
6.1.1	Transcript . . . . .	74
6.1.2	System additions . . . . .	77
6.2	Participant #2 . . . . .	80
6.2.1	Transcript . . . . .	81
6.2.2	System additions . . . . .	82
6.3	Participant #3 . . . . .	83
6.3.1	Transcript . . . . .	83
6.3.2	System additions . . . . .	85
6.4	Demo interaction . . . . .	87



<b>7</b>	<b>Discussion and conclusion</b>	<b>92</b>
7.1	Discussion . . . . .	93
7.1.1	Suitable for trivial question answering . . . . .	93
7.1.2	Systems can be built quickly . . . . .	94
7.1.3	Systems can be built by non-linguists . . . . .	94
7.1.4	Systems can be extended and maintained easily . . . .	94
7.1.5	Can engage in small talk . . . . .	95
7.1.6	Can respond to questions outside of its knowledge do- main . . . . .	95
7.1.7	Can engage in mixed-initiative dialogues, for non-trivial question answering . . . . .	96
7.1.8	Uses plans to help users achieve their information goals	96
7.1.9	Can deal with belief revision . . . . .	96
7.1.10	Outputs grounding dialogue moves . . . . .	97
7.1.11	Implements question accommodation . . . . .	97
7.2	Conclusion . . . . .	98
7.3	Further work . . . . .	98
7.3.1	Make domains easier to extend and maintain . . . . .	99
7.3.2	Add some plans for engaging in small talk . . . . .	99
7.3.3	Improve the suitable for trivial question answering . .	99
7.3.4	Rewrite parts of the system . . . . .	100
7.3.5	Allow for action-oriented dialogue . . . . .	100
7.3.6	Further ideas . . . . .	100
<b>A</b>	<b>statistics.pl</b>	<b>102</b>
<b>B</b>	<b>datetime.pl</b>	<b>113</b>
<b>C</b>	<b>timezonedb.pl</b>	<b>115</b>

# Chapter 1

## Introduction

### 1.1 Chatbots and dialogue management

A chatbot, or chatterbot[1], can be defined as a computer program that interacts with users using natural language[2], sometimes in a certain domain or on a certain topic[3], and possibly has an avatar and a speech processing module[4].

A growing number of chatbots are used on websites and instant messaging networks. An online directory of chatbots[5] currently lists 111 known chatbots for the Netherlands alone. These chatbots generally answer questions about products and services and are used on websites of companies such as the Nederlandse Energie Maatschappij[6], IKEA[7] and Action[8].

These systems function by matching user input with canned responses using regular expressions[9]. A canned response is a stored pre-written reply. The success of these systems depends upon having a large enough set of pattern/response pairs, since users may choose to converse about any topic.

#### 1.1.1 ELIZA

The origins of chatbots can be traced back to as early as the 1960's when Joseph Weizenbaum developed his ELIZA[10] program. ELIZA simulates a Rogerian psychologist and is capable of carrying on conversations like the following:

User: Men are all alike.

ELIZA: IN WHAT WAY

User: They're always bugging us about something or other.

ELIZA: CAN YOU THINK OF A SPECIFIC EXAMPLE

User: Well, my boyfriend made me come here.

ELIZA: YOUR BOYFRIEND MADE YOU COME HERE

User: He says I'm depressed much of the time.

ELIZA: I AM SORRY TO HEAR YOU ARE DEPRESSED

Although this dialogue seems impressive, ELIZA works according to a very simple principle. It uses pattern matching and substitution to process the input it receives and translate it into a suitable output. ELIZA often applies two or more substitutions to the user input, the first one consists of swapping all instances of *my* to *your*, and *I'm* to *you are*. Subsequent substitutions match and replaces other patterns in the input. Here are some examples[11]:

```
* you are (depressed|sad) * => I AM SORRY TO HEAR YOU ARE \1
* all * => IN WHAT WAY
* always * => CAN YOU THINK OF A SPECIFIC EXAMPLE
```

ELIZA gives the impression of intelligence by repeatedly applying this read-substitute-print loop. Although ELIZA was written between 1964 and 1966 and can be considered somewhat dated, modern day chatbots operate using the same principles.

### 1.1.2 Research dialogue systems

The simple read-substitute-print loop as used by chatbots works well for trivial question answering and small talk. Most human dialogue however is more complex than that. A key component of research dialogue systems is the dialogue manager which is responsible for the state and the flow of the conversation allowing for true mixed-initiative conversations. Research dialogue systems on the other hand are generally hard to implement and only converse within some limited domain like travel planning[12] or telephone call routing[13].

### 1.1.3 Information state approach

One successful architecture for dialogue management is the information state approach[14, 15]. In the information state approach the dialogue manager is composed of a structure called the information state which keeps track of the state of the conversation and a set of update rules.

The information state keeps track of information such as questions that have been raised, answers that have been given, private and shared beliefs and who did the last utterance. The update rules determine the flow of the conversation and take care of updating the information state and selecting the next conversational move. An example of an update rule might be one that answers questions the user may have asked. It would check if there are any unanswered questions for which there is an answer in the set of private beliefs but not in the set of shared beliefs. The rule then selects an answer as the next conversational move causing the system to announce the answer to the user and moves the answer to the set of shared beliefs.

## 1.2 Problem statement

Both the chatbot approach and the information state approach as used by research dialogue systems have their advantages and disadvantages.

Chatbot systems work well for trivial question answering and small talk. They are easy to build and require no special knowledge of computational linguistics. Human dialogue however is too complex to be handled by a simple read-substitute-print loop, which becomes apparent when what to say next depends on anything more than the current user input. Chatbots aren't goal-oriented. In other words, they do not try to help users achieve a task, except resolving questions for which a direct answer is available.

Research dialogue systems that apply the information state approach to dialogue management are able to engage in true mixed-initiative dialogue. They can handle phenomena such as grounding and belief revision. These systems are built to help users achieve certain goals that extend simple question answering. However, they can only do so within a very limited domain like travel planning and telephone call routing and need to be told how to parse natural language into a semantic representation that the dialogue manager can understand. This makes these kinds of systems hard to build and maintain even for experts.

## 1.3 Aims of this project

The purpose of this project is to investigate how the two approaches can be combined to get the best of both worlds. The aim is to develop a chatbot where the read-substitute-print loop has been replaced by a dialogue manager.

Such a hybrid approach would make it possible to quickly build systems that besides trivial question answering and small talk can engage the user in more complex conversations. These more complex conversations would be driven by the dialogue manager which recognizes the goal of the user and uses a set of ready made plans and update rules to determine what to say next.

Like in some research dialogue systems, the dialogue manager would also include the ability to deal with grounding and belief revision, something that is not found in current chatbot systems.

The dialogue manager in this hybrid would work with natural language directly instead of some semantic representation. This would make developing and extending systems based on this approach much easier for non-linguists since it is no longer necessary to write code that translates natural language to a semantic representation and vice versa.

The use of general patterns to match user input would enable the system to converse about, or at the least properly respond to, topics that extend

the domain the system was built for.

## 1.4 Research questions

The main research question is:

- How can a chatbot be equipped with a dialogue manager?

Sub-questions are:

- How does a chatbot work?
- What are different ideas on how to implement a dialogue manager?
- What are advantages and disadvantages of each approach?
- How does one use the information state approach ?
- How will such a hybrid system perform?
- What are its limitations?

The first four sub-questions will be answered by reviewing literature.

## 1.5 Overall approach

The main research question will be answered by actually building a system that is a mixture of a chatbot and a research dialogue system. Most sub-questions can be answered by studying literature or evaluating the resulting system.

To incorporate a dialogue manager into a chatbot the read-substitute-print loop will need to be extended to something more similar to an architecture that research dialogue systems use.

Part of a typical dialogue system architecture is the natural language understanding module that parses natural language and extracts its overall meaning into a logic or frame based knowledge representation which is used internally by the dialogue manager.

Instead of making use of a special semantic representation, the hybrid system makes use of natural language. The natural language understanding module will be replaced by a much simpler component that will clean up the user input by correcting spelling errors, removing superfluous words and swapping pronouns from *Im* to *YOU ARE*, and *my* to *YOUR*. Also undoing ellipsis (leaving out text that is implicitly understood) will be the responsibility of the component. The idea is to reduce the user input into a somewhat more restricted form of English and let the dialogue manager work with that instead of some logic or frame based knowledge representation.

Besides cleaning up user input, this component also categorizes user input as being a question, assertion, command and so on. All this cleaning up and categorization of user input will be done using patterns similarly to how chatbots match and rewrite user input.

As written above, there are several ideas on how to implement a dialogue manager. One system, described in detail in section 3.5.2, which deals with so called issue-based dialogue will serve as the main inspiration on how to implement a dialogue manager. Issue-based dialogue refers to dialogue that people usually engage in to obtain information. The information state for this particular system includes slots for keeping track of which questions are under discussion, shared and private information, an agenda for the next dialogue move, the last utterance and speaker, and a plan that is being executed for obtaining a certain (information) goal. All facts and questions that the dialogue manager keeps track of will be stated in natural language as well. The dialogue manager will again use pattern matching to determine if a certain fact resolves an open question.

## 1.6 Thesis outline

Chapter 2 explains the concepts behind chatbots and provides examples of pattern/response pairs specified in AIML, an XML dialect for building chatbots. An overview of research dialogue systems is given in chapter 3 together with a discussion of different dialogue management approaches including the information state approach. The methodology, that is based on the goals described in section 1.3, on how to equip a chatbot with a dialogue manager and its implementation are the topics of chapter 4 and 5. Chapter 6 shows the transcripts and feedback from users after chatting with the resulting system. Finally, chapter 7 concludes this thesis with a discussion of the results.

## Chapter 2

# Chatbot technology and human deception

The sub-question that this chapter addresses is:

- How does a chatbot work?

The answer to this question is twofold. Firstly, one can explain the inner workings of a chatbot and explain how the chatbot engine produces a response given a certain user input. Secondly, we can discuss the effect the chatbot's response has on a human user. This second aspect explains why the way a chatbot works works. In a sense it tells us more about the inner workings of humans than of chatbots.

Because the basic principle behind chatbots has been explained in section 1.1.1 we will start with the second aspect. This chapter concludes however with a description of a chatbot called ALICE[16] and illustrates how this chatbot is programmed.

### 2.1 The imitation game

In 1950, Alan Turing published an essay[17] in which he poses the question whether machines can think. Because of the imprecision of the terms *machine* and *think* he suggests an empirical test, in which a computer's use of language would be the basis for determining if a machine can think.

In this test, or game, nowadays known as the Turing test, there are three participants. Two of them are human and one is a computer. One of the humans plays the role of an interrogator with the objective of determining which of the other two participants is the computer. He or she has to do so by asking a series of questions via a teletype. The task of the machine is to try to fool the interrogator into believing it is a person. The second participant also wants to convince the interrogator that he or she is human.

Because of the fact that effective use of language is intertwined with our cognitive abilities, Turing believed that using language as humans do is sufficient as an operational test for intelligence.

## 2.2 Some early systems

The suitability of the Turing test as a test for intelligence has been an ongoing topic of debate among AI researchers and philosophers[18]. As was discussed in section 1.1.1, the system ELIZA uses simple pattern matching to process input and transform it into a suitable output.

ELIZA's relevance to the Turing test is that many people who interacted with the program became convinced that it really understood them. This to great dismay of ELIZA's creator Joseph Weizenbaum who writes:

I had not realized that extremely short exposures to a relatively simple computer program could induce powerful delusional thinking in quite normal people[19].

In one particular anecdote[20] a business executive intended to contact a programmer via teletype but without realizing got ELIZA instead. The dialogue went on for a while, and in the end the business executive phoned the programmer angrily asking why he was being so 'snotty'.

Based on the anecdote above, one might argue that ELIZA has already passed the Turing test. The executive did mistake ELIZA for a human being, even so he wasn't trying to distinguish between a computer and a human so the specifications of the Turing test weren't met. Furthermore, the procedure wasn't repeated with a number of different interrogators.

Another classic program is PARRY, created by the psychiatrist Kenneth Colby[21]. While ELIZA simulates a Rogerian therapist, PARRY simulates a paranoid schizophrenic. Although more advanced than ELIZA, PARRY also works by utilizing pattern matching and canned responses. Here is an example interaction[22].

Psychiatrist: Good afternoon. My name is Doctor Jones.  
PARRY: Pleased to meet you.  
Psychiatrist: How are you?  
PARRY: All right I guess.  
Psychiatrist: Could you tell me your name?  
PARRY: Pat Smith  
Psychiatrist: How old are you?  
PARRY: 28.  
Psychiatrist: I thought you might have some problems.  
PARRY: I've been upset lately.  
Psychiatrist: By what?



PARRY: People get on my nerves sometimes.  
 Psychiatrist: Tell me more.  
 PARRY: Do you know anything about bookies?  
 Psychiatrist: No. What about bookies?  
 PARRY: A bookie didn't pay me once.  
 Psychiatrist: When did this happen?  
 PARRY: A couple of months ago.

Colby wrote the program to better understand the paranoid mind and Colby and his team report[23] that PARRY has passed an approximation of the Turing test. In their experiment five psychiatrists conducted interviews via teletype terminals. Sometimes they were conversing with PARRY and sometimes with a human paranoid. For the psychiatrists the aim was to determine whether there was a computer or a human on the other end of the line. In the total number of ten interviews five were correct and five were incorrect. It is important to note, that the experiment was an approximation of the Turing test because the interrogators only conversed with either a human or a computer instead of both simultaneously.

In the Turing test interrogators may choose to converse about any topic. However, as philosopher Jack Copeland notes[24], the characteristics of a paranoiac's conversation, as PARRY was simulating, are precisely such as to prevent this from happening. PARRY's conversational style was like that of real paranoiacs: evasive, obstructive, hostile, and persistently coming back to delusional themes.

In the same way it was the conversational style of a therapist that made ELIZA so convincing. The setting of a psychiatric interview provides an excuse for ELIZA to know almost nothing of the real world. Thus ELIZA escapes the need for common knowledge.

## 2.3 Trickery and the Loebner contest

The Loebner contest was first held on November 8, 1991 and is advertised as the first formal instantiation of the Turing test[25]. This annual competition between computer programs is held to identify the most 'human' computer and eventually to reward \$100,000 for the program that first passes an unrestricted Turing test. The Loebner contest has been criticized for rewarding tricks rather than furthering the field of Artificial Intelligence[26, 27]. One paper[1], describing one of the entries in the contest named CHATTERBOT, gives an overview of the kind of tricks such a program typically uses and notes:

Here we unashamedly describe some of the better tricks, confident in the belief that when someday a computer program *does*

pass the Turing test, it will use many of them, for the simple reason that people already use them.

The paper first exposes the main trick ELIZA uses, namely using questions to draw a conversation out of the user. The program mainly asks question without any real contribution from itself. Since most people love to talk about themselves, they are happy to believe the program is listening. The illusion of listening is amplified by including substrings of the user's input in the program's output.

User: You hate me.

ELIZA: Does it please you to believe that I hate you?

Also note the use of pronoun swapping. 'You' is turned into 'I'. As stated in the previous section, the use of Rogerian mode provides a perfect cover for never saying anything declarative.

It then continues to describe some of the tricks that originated in PARRY:

- **admitting ignorance**, unlike ELIZA, PARRY does sometimes say *I don't know* in response to a question.
- **changing the level of the conversation**, saying *Why do you ask that?* as a way of changing the subject.
- **rigidly continuing a previous topic**, PARRY includes small stories about the Mafia, and tries to tell these in a fixed order.
- **introducing new topics**, launching into a new story.

These tricks simulate paranoid behaviour well, but also work in regular conversation.

The CHATTERBOT program uses the tricks above but also adds some additional ones:

- **having many fragments of directed conversation ready**, much like the opening book of a chess playing machine.
- **making controversial statements** to drag the user into the program's conversation rather than letting the user direct the discourse.
- **humorous statements** also make the program seem more human.
- **sometimes agreeing with the user.**
- **excerpting usenet news.**
- **simulate typing speed** by including realistic delays between characters. Slowing down the conversation also gives the person fewer chances to make mistakes.

Programs such as PARRY, ELIZA and more modern programs such as the entries of the Loebner contest are referred to as chatbots. They use tricks as described in this section while their inner workings are based on little more than pattern matching and outputting canned text.

People don't find these programs intelligent just because of the tricks that are used, but because of something that Douglas Hofstadter calls the *Eliza Effect*[28] which refers to

the susceptibility of people to read far more understanding than is warranted into strings of symbols - especially words - strung together by computers.

The next section describes an open source chatbot called ALICE, which was the most 'human' entry in the 2000, 2001 and 2004 Loebner contests and illustrates what goes into developing a somewhat convincing chatbot.

## 2.4 ALICE

ALICE is a chatbot developed initially by Richard Wallace[16]. It is inspired by ELIZA and uses an XML Schema called AIML for specifying the pattern/response pairs.

The first edition of ALICE was implemented in 1995 in SETL, but was migrated to Java in 1998. After that more and more people started contributing to it and in January 2000 ALICE became first in the Loebner contest. Later the ALICE/AIML engine was also ported to C/C++[29], PHP[30] and several other languages[31] contributing to the popularity of ALICE and AIML.

In May 2002 the website [www.pandorabots.com](http://www.pandorabots.com) came online allowing internet users to develop and host their own chatbots based upon AIML for free[31]. It currently hosts more than 206.000 chatbots.

## 2.5 Elements of AIML

AIML stands for Artificial Intelligence Markup Language and is an XML dialect used for specifying patterns and responses.

Before pattern matching the user input is converted to uppercase, and special characters such as question marks and commas are removed.

### 2.5.1 Categories

Categories are the basic units of knowledge in AIML. A category consists of at least a pattern and a template. A category element in AIML may look like

```
<category>
  <pattern>WHAT IS YOUR NAME</pattern>
  <template>My name is Francisco.</template>
</category>
```

and matches the input "What is your name?". The template is the response or reply. Patterns can also contain wildcards, which match one of more words, for example

```
<category>
  <pattern>WHAT IS YOUR *</pattern>
  <template>Guess!</template>
</category>
```

This pattern will match input such as: "What is your favorite color?", "What is your shoe size?", "What is your brother's name?", etc. The wildcard symbol \* will match one or more words while \_ matches a single word. Words consist of numbers and letters but no other characters. The pattern language is case invariant because the user input is first converted to upper case.

Often the template is as simple as some literal text like in the examples above. A template may use variables, such as the example

```
<category>
  <pattern>HOW OLD AM I</pattern>
  <template>You told me you are <get name="user-age" />
    years old.</template>
</category>
```

## 2.5.2 Recursion

A template may also contain the <srai> tag to recursively match other categories such as in

```
<category>
  <pattern>DO YOU KNOW WHO * IS</pattern>
  <template><srai>WHO IS <star /></srai></template>
</category>
```

This category makes sure that the input "Do you know who Socrates is?" will be redirected to a category that matches the input "who is socrates".

Simplifying complex grammatical forms into simpler ones is one application of the <srai> tag. Another use is handling synonyms.

```
<category>
  <pattern>HELLO</pattern>
  <template>Hi there!</template>
</category>
```

```

<category>
  <pattern>HI</pattern>
  <template><srai>HELLO</srai></template>
</category>
<category>
  <pattern>HOWDY</pattern>
  <template><srai>HELLO</srai></template>
</category>
<category>
  <pattern>HOLA</pattern>
  <template><srai>HELLO</srai></template>
</category>

```

This tag can also be used for catching and correcting grammatical errors, such as the common "your" when "you're" or "you are" is intended.

```

<category>
  <pattern>YOUR A *</pattern>
  <template>I think you mean "you're" or "you are"
    instead of "your".
    <srai>YOU ARE A <star></srai>
  </template>
</category>

```

### 2.5.3 Context

A category may also use the <that> to refer to the chatbot's previous utterance. The tag encloses a pattern that matches ALICE's previous output. A common application can be found in yes-no questions.

```

<category>
  <pattern>YES</pattern>
  <that>DO YOU LIKE MOVIES</that>
  <template>What is your favorite movie?</template>
</category>

```

This category is activated when a user answers "yes" to the question "Do you like movies?", and responds with "What is your favorite movie?".

An interesting use is a set of categories to respond to knock-knock jokes.

```

<category>
  <pattern>KNOCK KNOCK</pattern>
  <template>Who is there?</template>
</category>
<category>
  <pattern>*</pattern>
  <that>WHO IS THERE</that>

```

```

    <template><star /> who?</template>
</category>
<category>
    <pattern>*</pattern>
    <that>* WHO</that>
    <template>Ha ha, very funny.</template>
</category>

```

This would produce the following dialogue:

```

User:   Knock knock!
ALICE:  Who is there?
User:   Boo.
ALICE:  Boo who?
User:   Don't cry, it's only a knock knock joke.
ALICE:  Ha ha, very funny.

```

#### 2.5.4 Variables

AIML supports getting and setting variables with the `<get>` and `<set>` tags. The markup `<set name="user-name">Anna</set>` stores the string Anna under the variable named user-name. Subsequent activations of `<get name="user-name"/>` return Anna.

```

<category>
    <pattern>*</pattern>
    <that>WHAT IS YOUR NAME</that>
    <template>Nice to meet you, <set name="user-name"><
        star /></set>.</template>
</category>
<category>
    <pattern>WHAT IS MY NAME</pattern>
    <template>Your name is <get name="user-name" /></
        template>
</category>

```

#### 2.5.5 Pronoun swapping

The first substitution ELIZA always does is replace "you're" with "I'm", and "your" with "my" etc. ALICE also provides this function using the `<person>` tag. This tag most often operates directly on the `<star/>` tag. For that reason AIML defines the shortcut tag `<person/>` which is shorthand for `<person><star/></person>`. An example of this can be seen in the following category:

```

<category>

```

```
<pattern>MY MOTHER *</pattern>
<template>Who else in your family <person/>?</template>
</category>
```

This category would yield the following dialogue fragment:

User: My mother takes care of me.  
ALICE: Who else in your family takes care of you?

## Chapter 3

# Dialogue systems

The sub-questions that are addressed in this chapter are:

- What are different ideas on how to implement a dialogue manager?
- What are advantages and disadvantages of each approach?
- How does one use the information state approach?

Work on chatbot systems like ELIZA and PARRY had very little emphasis on the study of dialogue. The appearance of research dialogue systems awaited better understanding of human-human and human-computer dialogue. This chapter introduces the fundamental properties of human-human conversation and components of dialogue systems.

One component of a dialogue systems is the dialogue manager which is responsible for keeping track of the state and flow of a conversation. Different typologies of approaches to dialogue management exist, but the one described in section 3.4.5 classifies systems based on the dialogue phenomena that are handled.

The chapter concludes with a discussion of the information-state architecture[15] which can be seen as a generalization of the different approaches on dialogue management. It illustrates how the GoDiS[32] system is implemented based on this architecture by describing the data structures and some update rules that the system uses. The goal is to get a feeling of how the system handles some of the dialogue phenomena that are discussed in the beginnings of this chapter.

### 3.1 Speech acts

Oxford philosopher John Austin[33] worked out the key insight that an utterance in a dialogue is a kind of action being performed by the speaker. This idea is particularly clear in performative sentences such as:



I name this ship the Titanic.

When uttered by the proper authority it has the effect of changing the state of the world. In this case it causes the ship to have the name Titanic. In this sense it is similar to any other action. Austin called these kinds of actions *speech acts* and claimed that the utterance of any sentence can be analysed on three levels:

- **locutionary act**, the actual utterance of a sentence with a particular meaning
- **illocutionary act**, the act of asking, answering, promising, etc. in uttering a sentence
- **perlocutionary act**, the intentional effects upon the thoughts, feelings, or actions of the hearer

An utterance like "*You can't do that*" consists of a locutionary effect of expressing the proposition that something is not allowed. The illocutionary force might be protesting or warning, and the perlocutionary effect might be that of stopping the addressee from doing something.

The illocutionary act can according to John Searle be classified into one of five major classes[34]:

- **assertives**, committing the speaker to something being true: *suggesting, putting forward, swearing, boasting, concluding*
- **directives**, attempts by the speaker to get the addressee to do something: *asking, ordering, requesting, inviting, advising, begging*
- **commissives**, committing the speaker to some future action: *promising, planning, vowing, betting, opposing*
- **expressives**, expressing the psychological state of the speaker: *thanking, apologizing, welcoming, deploring*
- **declarations**, bringing about a different state of the world by an utterance: "*I quit*", "*You're fired*"

## 3.2 Grounding

In dialogue one cannot assume "perfect communication" in the sense that all utterances that are made are perceived and understood correctly. While conversing both speakers much constantly establish **common ground**[35]. Common ground refers to the set of things that are mutually believed by both speakers. To achieve common ground the hearer must **ground** the

speaker's utterances, making it clear that the speaker has understood the meaning and intention behind the utterance. Grounding is also important when the hearer has problems understanding the utterance and he or she has to indicate these problems so that mutual understanding can be achieved.

A hearer can use different methods for grounding the speaker's utterance. Five methods are [36]:

- **continued attention**, the hearer shows he or she is continuing to attend and by doing so remains satisfied with the speaker's presentation
- **next contribution**, the hearer starts in on the next relevant contribution
- **acknowledgement**, the hearer nods or says a continuer like *uh-huh*, *yeah* or an assesment like *that's great*
- **demonstration**, the hearer can demonstrate what has been understood by reformulating/paraphrasing or completing the speaker's utterance
- **display**, the hearer repeats verbatim all or part of the speaker's presentation

### 3.3 Implicature

During conversation there are many things that are never explicitly said but can implied. Consider the following dialogue fragment between a travel agent and a client:

Agent: And what day in May did you want to travel?

Client: I need to be there for a meeting that's from the 12th to the 15th.

In the fragment above the client doesn't directly answer the question. The client merely states that he or she has a meeting at a certain time and expects the travel agent to be able to infer that he or she wants to travel by the 12th.

Herbert Paul Grice proposed that what enables hearers to draw these inferences is a set of general heuristics that guide the interpretation of utterances [37]. These heuristics can be separated into the following four maxims:

- **Maxim of Quantity**: Be exactly as informative as is required
  1. Make your contribution as informative as is required (for the current purposes of the exchange)
  2. Do not make your contribution more informative than is required

- **Maxim of Quality:** Try to make your contribution one that is true
  1. Do not say what you believe to be false
  2. Do not say that for which you lack adequate evidence
- **Maxim of Relevance:** Be relevant
- **Maxim of Manner:** Be perspicuous
  1. Avoid obscurity of expression
  2. Avoid ambiguity
  3. Be brief (avoid unnecessary prolixity)
  4. Be orderly

Grice proposed that what enables a hearer to draw inferences is that they assume that a speaker is following these maxims. In the example fragment above, the maxim of relevance is what allows the agent to understand that the client only mentions the meeting because it was relevant at that point in the dialogue. The most natural inference is that the client meant the agent to understand that the departure time was before the meeting time.

A dialogue system can follow these maxims to make dialogue easier and more understandable. For example, a system should follow the maxim of quantity when a user asks for help and provide help with the task at hand and not provide information about all the things that the user can do with the system.

### 3.4 Dialogue system components

Figure 3.1 shows a typical architecture for a spoken dialogue system[11].

The speech recognition component takes audio input and returns a string of words. The natural language understanding unit extracts meaning from the string of words and produces a semantic representation. The natural language generation component converts meaning to text and the text-to-speech synthesis module maps this string of text to speech. The dialogue manager controls the whole process together with the task manager which has knowledge about the task domain.

The following sections describe these components in more detail.

#### 3.4.1 Speech recognition

The speech recognition component in a spoken dialogue system takes care of identifying spoken words. This is done by first translating an analogue signal to a digital, and then analyse the acoustic content of the signal. A

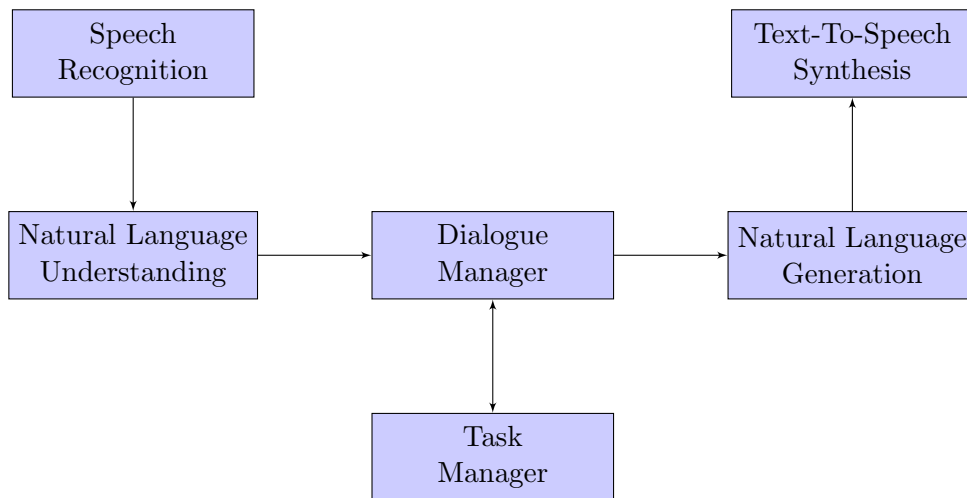


Figure 3.1: A spoken dialogue system architecture

word consists of one or more phonemes, and the speech recognition software tries to identify each word by identifying each phoneme in the signal. Each phoneme represents a sound, and can be described as "*the smallest contrastive linguistic unit which may bring about a change of meaning*" [38].

If a dialogue system can only be accessed using a text-only interface then it doesn't require a speech recognition component and user input is directly fed into the natural language understanding unit.

### 3.4.2 Text-to-speech synthesis

The text-to-speech synthesis component is responsible for transforming text into synthetic speech. One common method of creating speech electronically is the concatenation approach in which prerecorded human speech is merged together into new utterances. The prerecorded units can be of different length, such as phrases, words or phonemes [39].

The text-to-speech synthesis component is optional like the speech recognition component.

### 3.4.3 Natural language understanding

The natural language understanding component takes the textual user input and maps it to a semantic representation that the dialogue manager can understand. Common representations are based on a frame-and-slot semantics. A travel system which has the goal of helping a user find an appropriate flight would have a frame with slots for information about the flight. User input such as the sentence *Show me morning flights from Boston to San Francisco on Tuesday* might correspond with the following filled-out

frame[40]:

SHOW:

FLIGHTS:

ORIGIN:

CITY: Boston

DATE:

DATE-OF-WEEK: Tuesday

TIME:

PART-OF-DAY: morning

DEST:

CITY: San Francisco

One way a natural language understanding component could compose this semantic representation is by making use of a semantic grammar. A semantic grammar is a context free grammar where the left-hand sides of the rules correspond to the semantic entities being expressed, as in the following fragment:

$\langle \text{show} \rangle$	$\models$	show me   i want   can i see   ...
$\langle \text{depart\_time\_range} \rangle$	$\models$	[ after   around   before ] $\langle \text{hour} \rangle$   morning   afternoon   evening
$\langle \text{hour} \rangle$	$\models$	one   two   three   four   ...   twelve [ $\langle \text{ampm} \rangle$ ]
$\langle \text{flights} \rangle$	$\models$	[ a ] flight   flights
$\langle \text{ampm} \rangle$	$\models$	am   pm
$\langle \text{origin} \rangle$	$\models$	from $\langle \text{city} \rangle$
$\langle \text{destination} \rangle$	$\models$	to $\langle \text{city} \rangle$
$\langle \text{city} \rangle$	$\models$	Boston   San Francisco   Denver   Washington   ...

The result of a parse is the labeling of the input string with the semantic node labels:

$\langle \text{show} \rangle$	$\langle \text{flights} \rangle$	$\langle \text{origin} \rangle$	$\langle \text{destination} \rangle$	$\langle \text{depart\_date} \rangle$	$\langle \text{depart\_time} \rangle$
		from $\langle \text{city} \rangle$	to $\langle \text{city} \rangle$		
Show me	flights	from Boston	to San Francisco	on Tuesday	morning

The slots in the frame can be read almost directly off the resulting parse above perhaps after putting the fillers into some canonical form, like normalizing dates into a DD:MM:YYYY format and times into 24-hours time.

Semantic grammars are widely used but cannot deal with ambiguity and have to be created by hand which can be slow and expensive. Ambiguity can

be addressed by adding probabilities to the grammar as is done in the TINA system[41]. Another alternative that also avoids hand-coding of grammars is to use a model based on Hidden Markov Models[42, 40, 43, 44].

### 3.4.4 Natural language generation

The objective of the natural language generation component is to take a meaning representation of what to say from the dialogue manager and transform this into a natural language string. This can be done in two ways[45]. The simplest and most common method is template-based generation in which words are filled in slots of templates, for example:

Flights from  $\langle \text{src} \rangle$  to  $\langle \text{dest} \rangle$  on  $\langle \text{date} \rangle$ . One moment please.

The advantages of this method are that it's conceptually simple and tailored to the domain so often of good quality. The disadvantages are that it lacks generality and variation in style.

An alternative method is to make use of a rule-based generator[46]. Such a generator generally consists of three components, a **sentence planner**, a **surface realizer**, and a **prosody assigner**.

The sentence planner determines what words and syntactic structures will be used for expressing the content. It also determines what elements can be grouped together for more natural-sounding succinct output. The surface realizer combines everything into a syntactically and morphologically correct sentence. Finally the prosody assigner annotates the text with rhythm, stress and intonation markers for use by the speech synthesizer.

### 3.4.5 Dialogue manager

The dialogue manager is the component in a dialogue system that is responsible for the state and the flow of the conversation and coordinates the activity of the subcomponents in a dialogue system.

A dialogue manager can be implemented in various ways, and different typologies exist. One such categorization[47] discerns finite state-based systems, frame-based systems and agent-based systems. A more fine-grained categorization[48] classifies systems as finite state-based, frame-based, sets of contexts, plan-based models and agent-based. These categorizations classify systems based on the dialogue phenomena handled and go from least to most complex.

#### Finite state- (or graph-) based systems

In these systems the flow of the dialogue is specified as a set of dialogue states with transitions denoting various alternative paths through a dialogue graph. At each state the system produces prompts, recognizes (or rejects)

specific words and phrases in response to the prompt, and produces actions based on the recognized response.

The following is an example of an interaction with a basic finite state-based system in which the system verifies the user's input at each state of the dialogue:

System: What is your destination?  
User: London.  
System: Was that London?  
User: Yes.  
System: What day do you want to travel?  
User: Friday.  
System: Was that Sunday?  
User: No.  
System: What day do you want to travel?

The dialogue states and their transitions must be designed in advance. The previous interaction could be controlled by a finite state automaton like the one in figure 3.2.

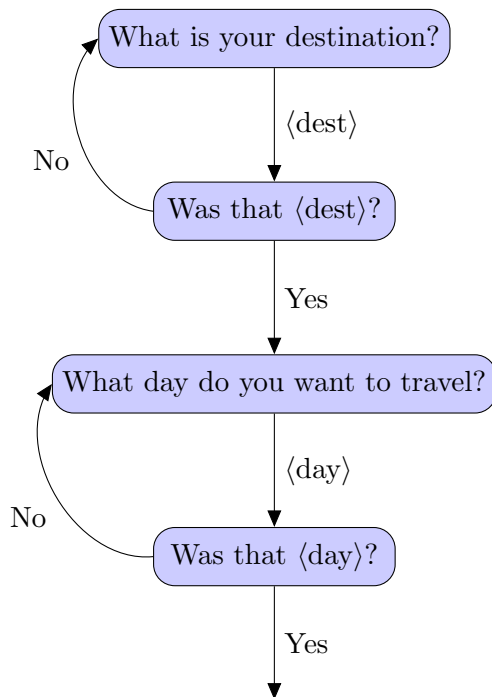


Figure 3.2: A partial finite-state automaton architecture for a dialogue manager

An example of a system using this approach is the Nuance automatic banking system[47].

## Frame-based systems

Frame- (or template-) based systems ask the user questions that enable the system to fill slots in a template in order to perform a task such as providing train timetable information. In this type of system the dialogue flow isn't fixed but depends on the content of the user input and the information that the system has to elicit.

Consider the following example:

System: What is your destination?  
User: London.  
System: What day do you want to travel?  
User: Friday.

If the user provides one item of information at a time then the system performs rather like a state-based system. In a frame-based system the user may provide more than the requested information.

System: What is your destination?  
User: London on Friday around 10 in the morning.  
System: I have the following connection...

In the second example, the system accepts the extra information and checks if any additional items of information are required before querying the database for a connection.

The context is fixed in these systems for they only do one thing. The context can be seen as being represented as a set of parameters that need to be set before the system action can be executed. For example, before a system can provide information about train arrivals and departures, the systems needs to know parameters like the day of travel, time of departure, and so on. The action is performed as soon as enough information has been gathered.

This approach has been used in systems that provide information about movies[49], train schedules[50] and the weather[51]. The advantage of the simplicity of these domains is that it is possible to build very robust dialogue systems. One doesn't need to obtain full linguistic analyses of the user input. For example, given the utterance *When does the Niagara Bullet leave Rochester?* the parameters ⟨train⟩ (The Niagara Bullet), ⟨event⟩ (leaving), and ⟨location⟩ (Rochester) can easily be extracted using simple pattern-matching.

## Sets of contexts

The frame-based approach can be extended with the concept of multiple contexts. For example, a simple travel booking agent may be used to book



a series of travel segments. Each travel segment can be represented by a context where each context holds the information about one travel leg and is represented using the frame-based approach. With multiple contexts the system is able to recognize when the user switches context. It may also be able to identify cases where a user goes back to modify a previously discussed context. An example of a system that uses this approach is the DARPA Communicator project[12].

### **Plan-based models**

Plan-based approaches are based on the view that humans communicate to achieve goals. These goals include the objective of changing the mental state of the listener. Plan-based theories of communicative action and dialogue[52, 53, 54] claim that the user input is not just to be seen as a sequence of words but as performing speech acts to achieve certain goals. The task of the listener is to infer and appropriately respond to the speaker's underlying plan.

For example, in response to a customer's question of "*Where are the steaks you advertised?*", the butcher might reply with "*How many do you want?*". This is an appropriate response because the butcher recognizes the underlying plan of the customer[55].

A system that is based on this approach is TRIPS[56] which in turn is based on TRAINS[57].

### **Agent-based systems**

In agent-based systems a conversation is seen as an interaction between two agents. Each agent is capable of reasoning about its own actions and beliefs, and about the actions and beliefs of the other agent.

These systems tend to be mixed initiative, which means that both the user and the system can take control of the dialogue by introducing new topics. Both agents work together to achieve mutual understanding of the dialogue, using discourse phenomena such as confirmation and clarification.

Agent-based systems focus on collaboration and are therefore able to deal with more complex dialogues that involve problem solving, negotiation, and so on. However, the approach does require more computing resources and processing power than other approaches.

Agent-based systems may use the Beliefs-Desires-Intentions (BDI) agent architecture[58] to model its internal state. The BDI-model has been extended for use in dialogue systems to also model mutual beliefs[59], i.e. what both agents belief to be true.

### 3.4.6 Task manager

The task manager is the component that has knowledge about the task domain, such as air travel. This component may also interface with external systems such as databases or an airline reservation system.

## 3.5 The information state approach

Section 3.4.5 describes various approaches to dialogue management. The information state approach[14] generalizes these approaches by viewing dialogue management in terms of information state updates.

Essential to this approach is the identification of relevant aspects of information in dialogue, how this information is updated and how these updating processes are controlled.

The term *information state* refers to the information about the dialogue that is stored internally by a dialogue system. This would be information such as questions that have been raised, answers that have been given, private and shared beliefs and who did the last utterance.

A set of rules, each consisting of a precondition list and an effect list, may change this information state and perform dialogue moves. If the preconditions of a rule match the information state the effect list is applied to the information state and one or more dialogue moves may be performed. Optionally there is a control algorithm that determines when which rules may activate from the applicable ones.

The information state approach allows us to compare a range of specific theories of dialogue management within the same framework.

### 3.5.1 TrindiKit

TrindiKit[60] is a toolkit written in SICStus Prolog that allows system designers to build dialogue management components according to their particular theories of information states.

Figure 3.3 shows an overview of the different components of TrindiKit.

A TrindiKit system consists of a number of modules for speech recognition, text-to-speech synthesis, natural language understanding and natural language generation. All these modules can read and update the information state using update rules. A controller ties the modules together which makes these modules independent of each other so that one module can be replaced without having to modify other modules. It is possible to connect databases or other external resources to the information state.

The *dialogue move engine* (DME) is the set of modules that is used to update the information state based on dialogue moves that have been observed, and it selects the next move to be performed depending on the current information state. The *total information state* (TIS) consists of

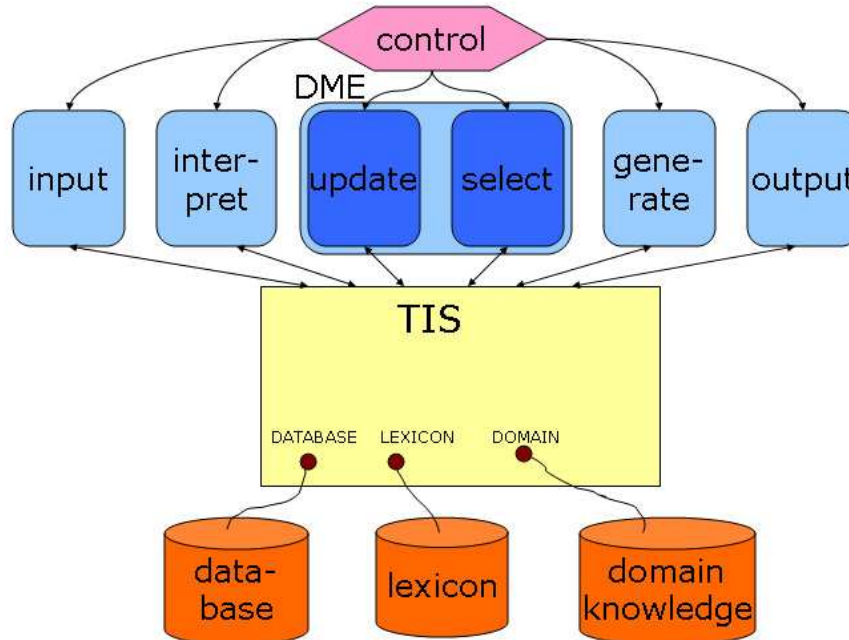


Figure 3.3: The TrindiKit architecture

the information state, and module and resource interface variables. Modules need interface variables that are designated parts of the TIS where the modules are allowed to read and write according to their associated TIS access restrictions. A *control* algorithm wires the modules together either serially or asynchronously. *Resources* can be databases, plan libraries, a lexicon and device interfaces that interact with external devices. Resources need interface variables to be accessible from modules.

To implement a specific dialogue theory using the TrindiKit system developers must specify

1. a description of the information state, that is the informational components of the theory of dialogue modeling including aspects of common context as well as internal motivating factors. These may include
  - participants common ground
  - obligations
  - commitments
  - beliefs
  - intentions
  - user models

2. a set of dialogue moves that will trigger the update of the information state. A dialogue move can be seen as an abstract description of an utterance and is specified as a speech act type (the illocutionary force) together with its content, for example *assert(destination(paris))* or *ask( $\lambda x$ .destination(x))*
3. a set of rules. Two kinds of rules can be discriminated namely update and selection rules. Update rules govern updating of the information state given various conditions of the current information state and performed dialogue moves. Selection rules are responsible for choosing a particular dialogue move to perform given conditions of the current information state. This enables participating in the dialogue rather than just monitoring one.
4. an update strategy for deciding which rules to select at a given point from the set of applicable ones. One strategy might be to pick the first rule that applies, but more sophisticated arbitration mechanisms are often used.

Some of the systems that have been developed using the TrindiKit are MIDAS[61], EDIS[62, 63], SRI Autoroute[64] and GoDiS[32, 65]. Some of these systems have been extended such as EDIS which forms the basis for a tutoring system called BEETLE[66] and GoDiS which is used in the D’Homme project[67] to control home appliances, in DICO[68] and DICO II[69] to control in vehicle systems such as the radio and cellphone, and to control robots[70].

TrindiKit isn’t the only toolkit available for developing dialogue systems based on the information state approach. Alternatives are DIPPER[71] which follows the design of TrindiKit closely but doesn’t support the specification of update strategies and Midiki[72].

The next section describes GoDiS and its implementation in TrindiKit in more detail.

### 3.5.2 GoDiS

GoDiS (Gothenburg Dialogue System) is a prototype dialogue system developed at Göteborg University, and implements a theory of issue-based dialogue management[73].

This system was used to explore how issues (modeled semantically as questions) can be used as a basis for dialogue management. The idea behind issue-based dialogue management is that issues, or questions, model conversational goals.

Issue-based dialogue management has elements of both frame-based and agent-based dialogue management.

### Information state

The information state is divided into two parts, *PRIVATE* and *SHARED*, see figure 3.4.

$$\begin{array}{l}
 \left[ \begin{array}{l}
 \textit{PRIVATE} : \left[ \begin{array}{l}
 \textit{AGENDA} : \textit{OpenQueue}(\textit{Action}) \\
 \textit{PLAN} : \textit{OpenStack}(\textit{PlanConstruct}) \\
 \textit{BEL} : \textit{Set}(\textit{Prop}) \\
 \textit{TMP} : \left[ \begin{array}{l} \textit{USR} : \textit{Tmp} \\ \textit{SYS} : \textit{Tmp} \end{array} \right] \\
 \textit{NIM} : \textit{OpenQueue}(\textit{Pair}(\textit{DP}, \textit{Move}))
 \end{array} \right] \\
 \\
 \textit{SHARED} : \left[ \begin{array}{l}
 \textit{COM} : \textit{Set}(\textit{Prop}) \\
 \textit{ISSUES} : \textit{OpenStack}(\textit{Question}) \\
 \textit{ACTIONS} : \textit{OpenStack}(\textit{Action}) \\
 \textit{QUD} : \textit{OpenStack}(\textit{Question}) \\
 \textit{PM} : \textit{OpenQueue}(\textit{Move}) \\
 \textit{LU} : \left[ \begin{array}{l} \textit{SPEAKER} : \textit{Participant} \\ \textit{MOVES} : \textit{Set}(\textit{Move}) \end{array} \right]
 \end{array} \right]
 \end{array} \right] \\
 \\
 \textit{Tmp} = \left[ \begin{array}{l}
 \textit{COM} : \textit{Set}(\textit{Prop}) \\
 \textit{ISSUES} : \textit{OpenStack}(\textit{Question}) \\
 \textit{ACTIONS} : \textit{OpenStack}(\textit{Action}) \\
 \textit{QUD} : \textit{OpenStack}(\textit{Question}) \\
 \textit{AGENDA} : \textit{OpenQueue}(\textit{Action}) \\
 \textit{PLAN} : \textit{OpenStack}(\textit{PlanConstruct})
 \end{array} \right]
 \end{array}$$

Figure 3.4: The GoDiS information state

The *PRIVATE* field consists of information which is private to the system. Its elements are

- an *AGENDA* that is read by the selection rules to determine the next dialogue move to be performed by the system.
- a *PLAN* which is a stack of actions. Some of the possible actions are findout, raise and respond.
- a set of propositions, *BEL*, that is used to store the things the system knows and may or may not have shared with the user.
- a *TMP* field which is used for grounding purposes.
- the *NIM* field which stands for Non-Integrated Moves. It is a structure to keep track of which dialogue moves have been interpreted so far.

The *SHARED* field contains the system's view of the mutual beliefs. Its elements are

- *COM* that contains the set of the propositions that the user and the system share.
- *ISSUES* which are topics that have been raised but not yet resolved.
- the *ACTIONS* that have been requested but not yet completed.
- a stack of questions under discussion, *QUD*. These are questions that are available for resolution using short answers, such as *yes*, *no*, and *to paris*.
- *PM*, previous moves, the moves that were made in the previous utterance.
- the field *LU*, which stands for latest utterance, holds the information about the latest utterance. It consists of two elements, namely:
  - *SPEAKER* which tells who the latest speaker was.
  - *MOVES* holding which moves were realized by the speaker's utterance.

Some examples of rules that update the information state can be seen in the remainder of this section. These rules are taken directly from the GoDiS source code<sup>1</sup>. The update rules that are detailed in this report are:

- `integrateUsrAsk`
- `integrateUsrAnswer`
- `accommodateIssues2QUD`
- `accommodatePlan2Issues`
- `accommodateDependentIssue`
- `clarifyDependentIssue`
- `accommodateCom2Issues`
- `retract`

Besides the information state, the TIS also contains the module interface variables that are used for interfacing the different modules, such as the modules that interpret and generate natural language. These are:

- *input* holding the user's utterances

---

<sup>1</sup><http://www.ling.gu.se/grupper/dialoglab/godis/>

- *latest\_moves* that lists the dialogue moves that are currently under consideration
- *latest\_speaker* holding the speaker that made the latest utterance
- *next\_moves* that lists the dialogue moves that the system will perform next
- *output* containing the system's utterances
- *program\_state* which is used to control the current state of the DME

### Dialogue moves

GoDiS uses the following seven dialogue move types:

- ask(q), where q is a question
- answer(p), where p is either a short answer or a proposition
- request(a), where a is an action
- confirm(a), where a is an action
- greet
- quit
- Interactive Communication Management (ICM) moves. These moves are explicit signals, formalised as dialogue moves, that are used for grounding.

### Resources

GoDiS includes resources that implement a travel agent that can answer questions about ticket prices and visa requirements. These resources include a set of plans, a lexicon and a database.

There are two plans, one to answer inquiries about ticket prices, and one about visa requirements.

```
plan( X^price(X),
  [ findout(X1^(how(X1))),
    findout(X2^(dest_city(X2))),
    findout(X3^(dept_city(X3))),
    findout(X4^(month(X4))),
    findout(X9^(dept_day(X9))),
    findout(X5^(class(X5))),
    consultDB(X7^(price(X7)))
  ] ).
```

```

plan( need_visa ,
      [ findout(X^dest_city(X)) ,
        findout(X1^citizenship(X1)) ,
        consultDB( need_visa )
      ] ).

```

When a user utters "*Price info please*" the system recognizes this as an inquiry about ticket prices. It sees that it has a plan for handling this kind of question and loads the plan into the *PLAN* field in the information state. The first action in this plan is to find out how the user wants to travel, that is, if the user wants to travel by plane, boat or train. After the system moves this first item of the plan into the *AGENDA* field, it outputs "*How do you want to travel?*".

The lexicon resource defines what input and output forms the system can recognize and produce. User input is interpreted using keyword spotting with clauses like these:

```

input_form( [ visa ] , ask(need_visa) ).
input_form( [ price ] , ask(X^price(X)) ).

input_form( [ by|S ] , answer(how(C)) ) :-
    lexsem( S, C), sem_sort(C, means_of_transport ).
input_form( [ in|S ] , answer(month(C)) ) :-
    lexsem( S, C), sem_sort( C, month ).

```

When a user says "*Do I need a visa?*", then the first clause above spots the keyword visa and interprets the user input as the dialogue move ask(need\_visa). The answer "*by plane*" to the question "*How do you want to travel?*" is interpreted as answer(how(plane)) by the third clause. The input is also interpreted and restricted by the predicates lexsem/2 and sem\_sort/2 which are defined in the resource files as well.

The system transforms dialogue moves to natural language using the predicate output\_form/2.

```

output_form( answer(price(Price)) ,
             [ 'The_price_is_', Price , '_crowns.' ] ).

output_form( answer(need_visa) ,
             [ 'Yes,_you_need_a_Visa.' ] ).
output_form( answer(not(need_visa)) ,
             [ 'No,_you_dont_need_a_Visa.' ] ).

```

The plans include the action consultDB. As soon as this action is pushed onto the *AGENDA* the database is queried and the result of this query, a proposition, is added to the *BEL* field. After the system performs the



dialogue move answer with this proposition as its content, the proposition is copied to the *COM* field because it is now assumed shared knowledge.

### Resolvedness and relevance

For GoDiS to be able to determine if an answer as given by a user resolves an open question it expects the domain resource to define a *resolves*(A, Q) relation where Q is a question and A is a proposition or elliptical (short) answer.

For example, as described in Larsson’s PhD thesis[73], the content of a resolving answer to a wh-question ?X.dest\_city(X) about destination city is either a proposition of the form dest\_city(C) or an underspecified propositional content C, where C has the conceptual category of city. So, if paris is defined to be a city concept, both dest\_city(paris) (e.g. ”I want to go to Paris”) and paris(”Paris”) resolve ?X.dest\_city(X).

It might be the case that an answer is relevant to the question but doesn’t actually resolve the question. For this reason GoDiS also expects a *relevant*(A, Q) relation to be defined. By definition all resolving answers are also relevant. An example of an answer that is relevant but not resolving would be not(paris).

### Grounding

In practical dialogue systems[74], grounding is often reduced to verification of the system’s recognition of user utterances. Two ways of handling verification are often distinguished, as exemplified below:

- I understood you want to depart from Madrid. Is that correct? (**explicit feedback**)
- You leave from Madrid. Where are you arriving at? (**implicit feedback**)

Explicit feedback signals understanding but also a lack of confidence in the interpretation and tries to elicit a response regarding the correctness of the interpretation from the hearer. Implicit feedback signals confidence in understanding and does not try to elicit a response.

While these kinds of feedback are useful and important in dialogue systems, it covers only a small part of the spectrum of possible grounding feedback as they occur in human-human dialogue.

GoDiS implements a broader categorization of feedback and sequencing moves represented as Interactive Communication Management (ICM) moves[75].

The general notation for feedback ICM moves in GoDiS is:

- icm:L \* P{:Args}

where  $L$  is an action level,  $P$  is a polarity and  $Args$  are arguments:

- $L$ : action level
  - con: contact (*"Are you there?"*)
  - per: perception (*"I didn't hear anything from you", "I heard you say 'to Paris'"*)
  - sem: semantic understanding (*"I don't understand", "To Paris."*)
  - und: pragmatic understanding (*"I don't quite understand", "You want to know about price."*)
  - acc: acceptance/reaction (*"Sorry, I can't answer questions about connecting flights", "Okay."*)
- $P$ : polarity
  - pos: positive
  - neg: negative
  - int: interrogative (or checking)

- $Args$ : arguments

The arguments depend on the action level and are aspects of the utterance or move that is grounded.

- for per-level: *String*, the recognized string
- for sem-level: *Move*, a move interpreted from the utterance
- for und-level:  $DP * C$ , where
  - \*  $DP$ : Participant is the dialogue participant who performed the utterance
  - \*  $C$ : Proposition is the propositional content of the utterance
- for acc-level:  $C$  Proposition, the content of the utterance

For example, the ICM move `icm:und*pos:usr*dest_city(paris)` provides positive feedback regarding the user's utterance that has been understood as meaning that the user wants to go to Paris.

Using this notation, GoDiS is able to give explicit feedback such as `icm:sem*pos` together with `icm:und*neg` indicating semantic, but not pragmatic understanding: *"I know the words that you said but I do not understand what you meant"*.

It can also produce different feedback depending on how high the recognition score is. This recognition score is calculated by the speech recognizer if the dialogue system is equipped with one. Consider the following rule:

```

rule( integrateUsrAsk ,
  [ $/private/nim/fst = M,
    M/fst == usr ,
    M/snd = ask(Q) ,
    $domain::plan( Q, - ) ],
  [ pop( /private/nim ),
    push( /private/agenda, icm:acc*pos ),
    ! $score = Score ,
    if_then_else( Score <= 0.7 ,
      push( /private/agenda, icm:und*int:usr*issue(Q) ),
      [ add( /shared/lu/moves, ask(Q) ),
        if_do( Score <= 0.9 ,
          push( /private/agenda, icm:und*pos:usr*issue(Q) ),
          % reraising of open issue
          if_do( in( $/shared/issues, Q ) and
            not fst( $/shared/issues, Q ),
            push( /private/agenda, icm:reraise:Q ) ),
          % reraising of closed issue
          if_do( in( $/shared/com, P ) and
            $domain::resolves( P, Q ),
            [ del( /shared/com, P ),
              if_do( in( $/private/bel, P ),
                del( /private/bel, P ) ),
              push( /private/agenda, icm:reraise:Q ) ] ),
            push( /shared/issues, Q ),
            push( /shared/qud, Q ),
            push( /private/agenda, respond(Q) ),
            add( /shared/lu/moves, ask(Q) )
          ] )
      ] ).

```

This rule checks the recognition score of the utterance and if it is less than or equal to 0.7, the rule pushes `icm:und*int:usr*issue(Q)` on the agenda causing the system to use explicit feedback to verify the system's recognition of user utterances. If the score is above 0.7 but still less than or equal to 0.9, the system will give implicit feedback. If the recognition score is above 0.9 no grounding feedback is produced.

The rule above may also push the move `icm:reraise:Q` onto the *AGENDA*. This ICM move is an example of a sequencing move. Sequencing moves are used to keep the user informed about what's going on inside the system.

The three types of sequencing ICM that GoDiS implements indicate:

- the loading of a plan.  
`icm:loadplan ("Let's see.")`

- an issue being reraised, in contrast to being raised for the first time, either implicitly or explicitly  
 icm:reraise ("So,...")  
 icm:reraise:Q ("Returning to the issue of Price.")
- some inference has been performed and that this has been successful  
 icm:accommodate ("Alright.")  
 icm:accommodate:Q (Alright, you want to know about ...")

### Question accommodation

Question accommodation[32] allows a user to divert from the fixed sequence of questions stated in a plan and allows the system to correctly interpret elliptical answers. An elliptical answer is an answer that requires the question in order to be understandable. For example, the answer "*to Paris*" is understandable only if you know that the question was "*Where do you want to go to?*".

There are three main types of question accommodation. Next to these three types we can also discern a fourth type which is question re-accommodation:

- **Local accommodation** allows a user to give elliptical answers on questions that have been raised earlier. When a user gives an elliptical answer the system checks if this short answer is relevant to the question that is on the *QUD* stack. The rule `integrateUsrAnswer` takes care of local accommodation.

```
rule( integrateUsrAnswer ,
[ fst( $/private/nim, DPM ),
  DPM/fst == usr ,
  DPM/snd = answer(A) ,
  % RELEVANCE
  fst( $/shared/issues , Q ),
  $domain :: relevant( A, Q ),
  not( ( not $domain :: proposition( A ) ) and
        not in( $/shared/qud, Q ) ),
  $domain :: combine( Q, A, P ),
  % ACCEPTABILITY
  $database :: validDBparameter( P ) or P = ( not _ )
],
[ pop( /private/nim ),
  if_then_else(
    in( $latest_moves , answer(A) ),
    !$score = Score ,
    ! Score = 0.6 ),
  if_then_else( Score ==< 0.7 ,
```

```

push( /private/agenda, icm:und*int:usr*P ),
[ add( /shared/com, P ),
  add( /shared/lu/moves, answer(P) ),
  if_do( not in( $/private/agenda, icm:acc*pos ),
    push( /private/agenda, icm:acc*pos ) ),
  if_do( Score <= 0.9 and A \= yes and A \= no,
    push( /private/agenda, icm:und*pos:usr*P ) )
] )
] ).

```

If an utterance included an answer as a dialogue move and this answer is relevant to an open question on the *ISSUES* stack then the answer and question are combined and the resulting proposition is added to the common knowledge *COM*. The answer can be either a full proposition or an elliptical answer.

If it is an elliptical answer the rule expects the issue to also be on the *QUD* stack.

For example if answer(Paris) was one of the dialogue moves of the latest utterance, and X2^dest\_city(X2) is on the *QUD* stack then Paris and X2^dest\_city(X2) are combined into dest\_city(Paris). If the recognition score is higher than 0.7 the proposition is added to *COM*. If the recognition score is below 0.7 the rule pushes an ICM move onto the agenda causing the system to output explicit feedback.

The following rule makes sure that in the case of an elliptical answer the issue is copied to the *QUD*.

```

rule( accommodateIssues2QUD,
[ $/private/nim/element/snd = answer(A),
  $latest_speaker == usr,
  not $lexicon :: yn_answer(A),
  in( $/shared/issues, Q ),
  not in( $/shared/qud, Q ),
  $domain :: relevant( A, Q ),
  not $domain :: proposition( A )
],
[ push( /shared/qud, Q ),
  raise( /shared/issues, Q ) ] ).

```

The accommodateIssues2QUD rule executes when the user makes an answer move, that isn't on the *QUD* stack yet. Besides pushing the issue onto the *QUD* stack the rule also makes the issue topmost on the *ISSUES* stack.

- **Global accommodation** allows a user to answer questions that have

not yet been raised but are to be raised in the plan. This way, the user can give all the information necessary at once:

User: Do I need a visa to go to Paris? I'm from Sweden.  
GoDiS: Yes, you need a visa.

instead of:

User: Do I need a visa?  
GoDiS: Where do you want to go to?  
User: to Paris.  
GoDiS: Where are you from?  
User: I'm from Sweden.  
GoDiS: Yes, you need a visa.

The following rule takes care of global accommodation.

```
rule( accommodatePlan2Issues ,
  [ $/private/nim/elem/snd = answer(A) ,
    $latest_speaker == usr ,
    not $lexicon :: yn_answer(A) ,
    ( in( $/private/plan , findout(Q) ) or
      in( $/private/plan , raise(Q) ) or
      in( $/private/plan , bind(Q) ) ) ,
    $domain :: relevant( A, Q ) ,
    not in( $/shared/issues , Q ) ,
    $domain :: default_question( Q ) or
      ( not ( ( in( $/private/plan , findout(Q1) ) and
                  Q \= Q1 ) and
                $domain :: relevant( A, Q1 ) ) )
    ] ,
  [ push( /shared/issues , Q ) ] ).
```

The `accommodatePlanIssues` rule checks if the answer move that the user produced is relevant to one of the question actions in the currently loaded plan. If this is the case, the question is pushed onto the *ISSUES* stack after which the `integrateUsrAnswer` rule is executed. If the `accommodatePlan2Issues` rule finds more than one question on the plan to which the answer is relevant it only pushes the question onto the *ISSUES* stack when one of these questions is declared the default question in the domain resource.

- **Dependent accommodation** enables a user to give information without telling which issue he or she wants to resolve. If more than one plan fits the user's answers, the system asks which issue the user wants to resolve.

User: I want to go to London.  
GoDiS: Do you want visa information or know about price?  
User: Visa information please.  
GoDiS: Where are you from?  
User: I'm from Germany.  
GoDiS: You do not need a visa.

The rule `accommodateDependentIssue` handles dependent accommodation.

```
rule( accommodateDependentIssue ,
[ $latest_speaker == usr ,
  setof( A, $/private/nim/eleme/snd = answer(A), AnswerSet ),
  $$arity( AnswerSet ) > 0,
  is_empty( $/private/plan ),
  $domain :: plan( SuperQ, Plan ),
  forall( in( AnswerSet, A ),
    in( Plan, findout(Q) ) and
    $domain :: relevant( A, Q ) ),
  not ( ( $domain :: plan( SuperQ1, Plan1 ) and
    SuperQ1 \= SuperQ ) and
    forall( in( AnswerSet, A ),
      in( Plan1, findout(Q) ) and
      $domain :: relevant( A, Q ) ) ) ),
  not in( $/private/agenda, icm:und*int:usr*issue(SuperQ) ) ],
[ push( /shared/issues, SuperQ ),
  push( /private/agenda, icm:accommodate:SuperQ ),
  push( /private/agenda, icm:und*pos:usr*issue(SuperQ) ),
  set( /private/plan, Plan ),
  push( /private/agenda, icm:loadplan )
] ).
```

This rule checks whether the user has made some answer moves that are relevant to exactly one plan in the domain resource while there isn't an active plan loaded yet. If it encounters such a plan it makes the issue belonging to this plan topmost in the *ISSUES* field, loads the plan and pushes some ICM moves onto the *AGENDA* notifying the user about what's going on.

The following rule raises a clarification question in case multiple plans match:

```
rule( clarifyDependentIssue ,
[ in( $/private/nim, pair(usr,answer(A) ) ),
  is_empty( $/shared/issues ),
  setof( MQ,
```

```

$domain :: depends( MQ, Q ) and
$domain :: relevant( A, Q ), MQs ),
remove_unifiables( MQs, MQs1 ),
$$arity( MQs1 ) > 1 ],
[ ! setof( P, in( MQs1, MQ1 ) and (P=issue(MQ1)), Ps ),
  push( /private/agenda, findout(Ps) )
] ).

```

It utilizes the predicate `depends/2`. This predicate, which has to be defined in the domain resource, relates questions the system can ask to the issues the system can resolve. The rule constructs a set of dependent issues for which there is a plan containing an action to which the answer is relevant. If there is more than one issue in the set the rule pushes a `findout` action onto the *AGENDA*.

- **Question re-accommodation** gives the user the possibility to revise previously given information.

```

User:   I would like some price information.
GoDiS:  How would you like to travel?
User:   I'd like to go by train.
GoDiS:  Where do you want to go to?
User:   No, not by train, I want to go by plane.
GoDiS:  Okay. Where do you want to go to?

```

This works as follows. If the user gives an answer for which there is no open issue on the *ISSUES* stack the answer won't be integrated by the `integrateUsrAnswer` rule. Instead the following rule will execute.

```

rule( accommodateCom2Issues ,
  [ $/private/nim/elem/snd = answer(P) ,
    $latest_speaker == usr ,
    $domain :: proposition( P ),
    in( $/shared/com, P1 ),
    $domain :: question( Q ),
    $domain :: relevant( P, Q ),
    $domain :: relevant( P1, Q ),
    not in( $/shared/issues, Q ) ],
  [ push( /shared/issues, Q ) ] ).

```

This rule checks if the answer is a valid proposition that is relevant to a question to which a proposition in the *COM* field is also relevant. In other words, the user has answered a question for which an answer has been supplied already. The system assumes the user wants to revise some issue and pushes it onto the open *ISSUES* stack.



```

rule( retract ,
  [ $/private/nim/elem/snd = answer(A) ,
    in( $/shared/com, P1 ) ,
    fst( $/shared/issues , Q ) ,
    $domain :: relevant( P1, Q ) ,
    $domain :: relevant( A, Q ) ,
    $domain :: combine( Q, A, P ) ,
    $domain :: incompatible( P, P1 ) ] ,
  [ del( /shared/com, P1 ) ] ).

```

The retract rule subsequently executes and removes the old information from *COM* in case it is incompatible with the new answer. Now the integrateUsrAnswer rule will execute integrating the new answer by updating the *COM* field.

## Chapter 4

# Methods

The main research question is:

- How can a chatbot be equipped with a dialogue manager?

Section 3.5.2 in the previous chapter describes in detail how the dialogue system GoDiS uses an issue based approach to dialogue management. Chapter 5 shows by implementation how a chatbot system can be extended with this approach and by doing so answers the main research question.

The following two questions concern how well this hybrid system performs and what its shortcomings are.

- How will such a hybrid system perform?
- What are its limitations?

This chapter discusses and motivates the research methods that were selected to address these two sub-questions.

### 4.1 Features and properties

Section 1.3 states the expected features and properties of the hybrid prototype. In order to answer the two sub-question, these characteristics will be examined in more detail.

1. Suitable for trivial question answering.
2. Systems can be built quickly.
3. Systems can be built by non-linguists.
4. Systems can be extended and maintained easily.
5. Can engage in small-talk.

6. Can respond to questions outside of its knowledge domain.
7. Can engage in mixed-initiative dialogues, for non-trivial question answering.
8. Uses plans to help users achieve their information goals.
9. Can deal with belief revision.
10. Outputs grounding dialogue moves.
11. Implements question accommodation.

Characteristics 1-6 correspond to properties of chatbots, while characteristics 7-11 are features of GoDiS or derived dialogue systems.

Another possible categorization is by relevance for a stakeholder. Characteristics 2, 3, 4, and 7 are relevant for system developers, while characteristics 1, 5, 6, and 8 to 11 are mostly relevant for users of the system.

## 4.2 Case study

The research questions are open-ended and therefor a qualitative research design is most appropriate.

The characteristics as listed in the previous section were used as indicators of how well the system performs. From the characteristics could be concluded that there are two units of analysis. One is related to the user of the system, and the other one is related to the system developer. The first unit of analysis is the conversation between a user and the system. The second unit of analysis is the construction and maintenance of the dialogue system.

In order to determine the usefulness the system, a domain is implemented that can answer questions about statistics, more specifically, it can help a user determine which statistical technique should be used in which situation. For instance, after a user asks, "*Which statistical test should I use*", the system will ask questions like "*How many independent variables are there?*", and "*What is the level of measurement of the dependent variable*". If mid-dialogue the user doesn't understand the question, he or she might ask "*What is an independent variable?*", after which the system comes up with a definition, and allows the conversation to continue.

This domain makes a good use case for several reasons.

- Many students and researchers struggle with deciding when to apply which statistical procedure and would greatly benefit from a system that implements this domain. A system that has practical value makes it easy to find willing students and staff for testing purposes.

- This domain requires the system to answer both questions for which a direct answer exists, such as "*What is an independent variable?*", and questions that require some further interaction, like "*Which statistical test should I use?*" A hybrid system should be able to handle both kinds of questions equally well.
- Because the subject of statistics is vast, it is impossible the system will be able to answer every possible question a user might raise. This isn't the objective either, because we're also interested in how the system deals with questions it can't answer. Furthermore, every time the user asks a question that the system can't answer, we can explore how well new knowledge can be added to the system.
- Unlike other toy domains such as call forwarding or travel planning, this domain really pushes the hybrid prototype to its limits. This is needed to expose it's shortcomings.

### 4.3 Implementation

Section 5.6 describes in detail how the statistics domain is implemented. It will show exactly what is needed to add a new domain.

Generally three actions must be taken.

- Define one or more plans. A plan consists of a pattern defining the issue to resolve and a series of actions. A plan is needed when the system must be able to answer questions for which no direct answer is available. For example, a question such as "*Which statistical technique should I use?*" can only be answered by asking the user some questions first, like "*How many independent variables are there?*" and "*Does the data meet assumptions for parametric tests?*". A plan is needed when answering a question requires further conversation.
- Define some patterns that match with user input, resolve questions or rewrite output. These patterns are defined as so called *units*. There are three different kinds of units: interpret units, resolve units and generate units. The hybrid system already defines a set of units that can handle some English, but some additional units might be necessary. How many depends on the domain.
- Add some facts to the domain. These facts will be used as direct answers to questions.

While taking these three actions we'll get an idea of how much time and effort is needed to get a minimal system running.

## 4.4 Procedure

In order to evaluate the system, three participants will converse with the system. The first participant will be a bachelor student, the second a master student and the third a PhD student. As can be expected, each participant will have a different level of knowledge of statistics.

After each interaction between a participant and the system, the conversation will be analysed to determine where the system wasn't able to adequately respond to the user's input. This analysis will indicate where the system needs improvement.

It's somewhat up to chance which, if any, linguistic phenomena such as question accommodation, information revision and grounding the user interactions will demonstrate. A lot of user interactions might be needed to show them all. For this reason a demo interaction will be added that specifically triggers the system to produce them.

## 4.5 Instructions

The following spoken instructions were given:

You are about to use a computer program that can understand and produce English. You can converse with this program by typing on the keyboard and reading from the screen. Perhaps you've conversed with a chatbot before either on a website or through an instant messaging program such as MSN. This program is similar, but unlike other chatbots this program can answer questions about statistics and help you select the appropriate technique for answering quantitative research questions.

I will give you a research question and I want you to converse with the program to find out which statistical technique is most appropriate for answering this question. Perhaps you already know which statistical test can best be used. If this is the case, then I want you to test if the system will come up with the same answer as you have. Remember that we are not testing how much *you* know about statistics. You won't be graded. We are strictly interested in the program's ability to converse in English. If at any time during the conversation the program asks a question that you don't understand, then try asking the program the meaning of the word or phrase that you don't understand. Try to pretend that you're talking to an actual person that understands English the way you and I do. The conversation that you're about to have will be logged for further analysis.

After the program gives you an answer, I have some further instructions for you. Do you have any questions so far?

The participants were led to a computer where the program was running and were given a sheet of paper with the following text:

The research question is:

What is the relationship between gender and dropout rates from an MSc course? What you have:

- the variable **sex**, with possible values male/female
- the variable **dropout** with possible values Yes/No

The statistical technique to answer the research question should be *Chi-square test for independence*. After the participant is done interacting with the program the instructions continue.

I now want you to ask the computer program three questions. One statistics related question, and two personal questions. The statistics question can be anything from simple to advanced. The personal questions can be anything you would normally ask a person you've just met.

Notice that the participants weren't suggested any example questions or wordings. This was done to ensure that the participants would come up with their own ideas and not that of the researcher.

## 4.6 Evaluation

Evaluation of this hybrid approach takes place during three different moments, namely

- during implementation of the statistics domain
- during the users' interactions
- while extending the system

During implementation of the statistics domain we're interested in how easy or hard it is to build the initial implementation of the statistics tutor. In other words, what is needed to get a minimal system working.

During the user's interactions we expect the system to produce grounding feedback and correctly deal with question accommodation. Also feedback that indicates the system can't deal with specific user input is of interest because it demonstrates specific points of improvement. Feedback such as

*"I don't understand"* indicates that the system doesn't recognize any of the user input, while *"I don't know"* is uttered when the system can integrate a question but has no fact or plan to resolve it.

The user's interactions show points of improvement for the domain implementation. These points of improvement allow one to evaluate what goes into extending the system.

## Chapter 5

# Implementation

This chapter outlines the key design decisions that were made in creating the prototype. It also provides a comprehensive overview of the system. The contents of this chapter supply an answer to the main research question:

- How can a chatbot be equipped with a dialogue manager?

In essence, the program described in this chapter is a hybrid of a chatbot and a research dialogue system. The ALICE chatbot system as described in section 2.4 will be used as the main inspiration for the chatbot part. The GoDiS system which was described in section 3.5.2 is the reference dialogue system on how to implement a dialogue manager.

### 5.1 SWI-Prolog

The prototype is developed in the logic programming language Prolog. One advantage of Prolog over other programming languages is that the language has built in support for expressing grammars in a formalism called definite clause grammar (DCG). This is useful for processing natural languages which is something that Prolog was originally created for[76].

Here is a simple Prolog DCG:

```
sentence —> noun_phrase , verb_phrase .
noun_phrase —> determiner , noun .
noun_phrase —> pronoun .
verb_phrase —> transitive_verb , noun_phrase .
verb_phrase —> intransitive_verb .
determiner —> [the] .
noun —> [boy] ; [girl] ; [apple] ; [cat] ; [fish] .
pronoun —> [john] ; [mary] .
transitive_verb —> [ate] ; [saw] .
intransitive_verb —> [slept] ; [wept] .
```



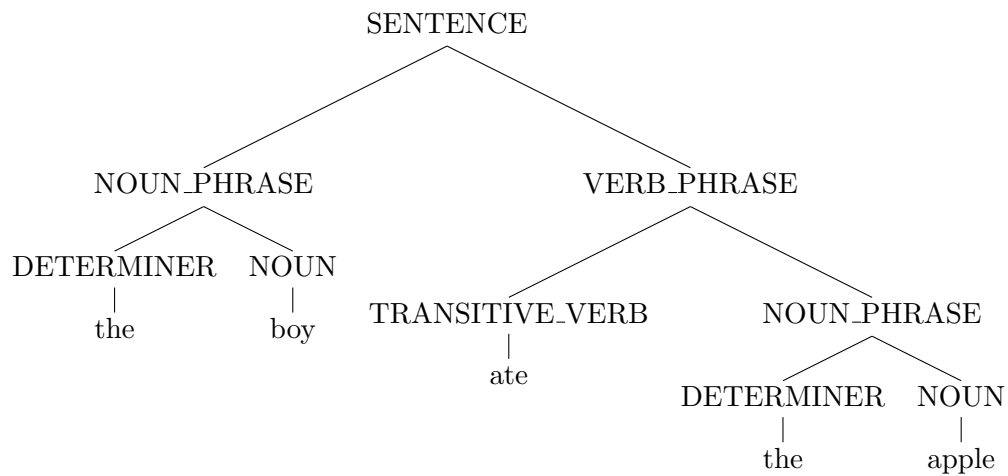


Figure 5.1: The boy ate the apple.

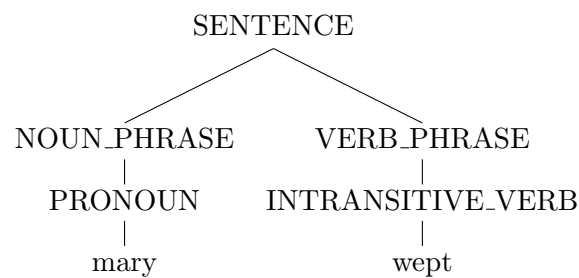


Figure 5.2: Mary wept.

This simple grammar accepts sentences such as: *The boy ate the apple* and *Mary wept*. Example parse trees can be seen in figure 5.1 and figure 5.2.

SWI-Prolog is used as the Prolog implementation of choice. Many implementations exist but the main reasons for choosing SWI-Prolog are:

- It's free and open source.
- It's fast, small and robust.
- It's portable and runs on all major platforms.
- It has many libraries including one for building web applications. This allows Prolog programs to be exposed as web services.

## 5.2 Pattern matching

As stated in section 2.4, many implementations for ALICE exist in a variety of programming languages. Lecturer Diego Mollá Aliod wrote a version for

Prolog which he uses in his *Language technology* course<sup>1</sup> at the Macquarie University, Australia.

Instead of using AIML, the program lets one specify categories as Prolog clauses. For example

```
category ([
    pattern ([what, is, your, name, '? ']) ,
    template ([ 'My', name, is, ' Francisco ', '. '])
]).
```

or

```
category ([
    pattern ([what, is, your, star(-), '? ']) ,
    template ([ 'Guess! '])
]).
```

which correspond to the first two AIML fragments in section 2.5.1.

The program also allows for recursive pattern matching, such as

```
category ([
    pattern ([do, you, know, who, star(A), is, '? ']) ,
    template ([ srai ([who, is, A, '? '])])
]).
```

which is adapted from the first example in section 2.5.2.

One nice feature of this ALICE implementation is that one can do pattern matching using DCG grammars. For instance

```
category ([
    pattern ([i, like, star(A), with, syntax(noun_phrase, B, [])]) ,
    template ([ 'Really?! ', 'I ', also, like, A, with, B, '. '])
]).
```

could yield the following interaction

User: I like playing with the cat

ALICE: Really?! I also like playing with the cat.

provided that the DCG grammar from section 5.1 has been specified.

The prototype will reuse some of the prolog predicates in the ALICE prolog program such as the one that handles pattern matching of user input.

## 5.3 Rule engine

TrindiKit, as introduced in section 3.5.1 and used by GoDiS, is a toolkit used for developing dialogue systems based on the information state approach.

---

<sup>1</sup><http://comp.mq.edu.au/units/comp248/index.html>

The toolkit itself is both free and open source for academic purposes but depends on the non-free SICStus prolog.

The core functionality of TrindiKit is that of a rule engine with the added ability to define update strategies. For the prototype it was decided to not use TrindiKit but another rule engine. This has been done for two reasons:

- The dependency on SICStus Prolog forces users of TrindiKit to use a commercial Prolog while excellent free and open source implementations exist.
- TrindiKit was designed to allow for comparison between different dialogue theories by making it as versatile and flexible as possible. The toolkit also includes modules for speech recognition, speech generation, and database and device interfacing. All this flexibility makes the toolkit somewhat hard to understand, use and configure.

The rule engine that is used is a stripped down adaptation of MIKE[77]. MIKE, which is an acronym for Micro Interpreter for Knowledge Engineering, was used in the British Open University's 'Knowledge Engineering' course. It's somewhat dated with its last update being version 2.50 in 1990, but it's open source, and runs on most Prolog implementations with some adjustments. MIKE supports knowledge representation using frames, and has a text based menu interface. Because the prototype won't be needing frames nor a menu interface, these extra functionalities have been stripped from MIKE leaving only a bare rule engine.

## 5.4 Information state

The information state used by GoDiS as described in section 3.5.2 and shown in figure 3.4 is highly structured with a clear separation between private and shared segments. This section describes the information state that is used in the prototype. It is mostly identical to the one in GoDiS with some simplifications. See figure 5.3 for details.

<i>AGENDA</i>	: <i>List(Action)</i>
<i>PLAN</i>	: <i>List(PlanConstruct)</i>
<i>BEL</i>	: <i>List(prop)</i>
<i>COM</i>	: <i>List(Prop)</i>
<i>QUD</i>	: <i>List(Question)</i>
<i>THAT</i>	: <i>List(Move)</i>

Figure 5.3: The information state for the hybrid chatbot/dialogue system

One of the most obvious changes in the information state in figure 5.3 compared to the one in figure 3.4 is the lack of hierarchy in the information

state structure. The fields in the information state have been reduced to simple prolog lists in the working memory of the rule engine.

Besides the information state, the rule engine also keeps track of variables **score**, **length**, **skipped** and a list **input**. The list **input** contains the user input that has not been matched yet. The variable **length** is set to the amount of words in the **input**. By keeping track of how many words have been skipped before the system found a match, a **score** is calculated that indicates what fraction of the user input has been matched by an input pattern. For details on how this is done, see the section labeled grounding under section 5.5.3.

What follows is a description of the fields that comprise the information state. Many of the fields have the same purpose as the ones in GoDiS.

- the *AGENDA* is read by the selection rules to determine the next dialogue move to be performed by the system.
- the field *PLAN* is a list of actions. Some of the possible actions are *findout*, *raise* and *respond*.
- the field *BEL*, is used to store the things the system knows and may or may not have shared with the user.
- *COM* that contains the set of the propositions that the user and the system share.
- a list of questions under discussion, *QUD*. Unlike the field with the same name in GoDiS, this field doesn't just contain the questions that are available for resolution using short answers. It's purpose is closer to the field *ISSUES* in GoDiS, that contains all questions that are open issues. The field *ISSUES* has been removed from the information state.
- *THAT* is a list to which elements are added each time the system interprets or generates text. The field contains a history of system and user utterances, and is queried when interpreting user input. For this reason it is named after the <that> AIML tag as described in section 2.5.3 which has a similar purpose.

Possible propositions that are stored in the *BEL* and *COM* fields are [ *paris, is, a, city* ], [ *the, sky, is, blue* ] and [ *my, name, is, pedrito* ]. It should be clear that all the things the system knows is not represented using some kind of artificial knowledge representation like first-order logic or frames, but as natural language. All reasoning, retrieval and manipulation of this knowledge is done using pattern matching and rewriting.

## 5.5 System architecture

Figure 5.4 depicts the overall architecture of the prototype, which largely overlaps with the typical dialogue system architecture as seen in figure 3.1.

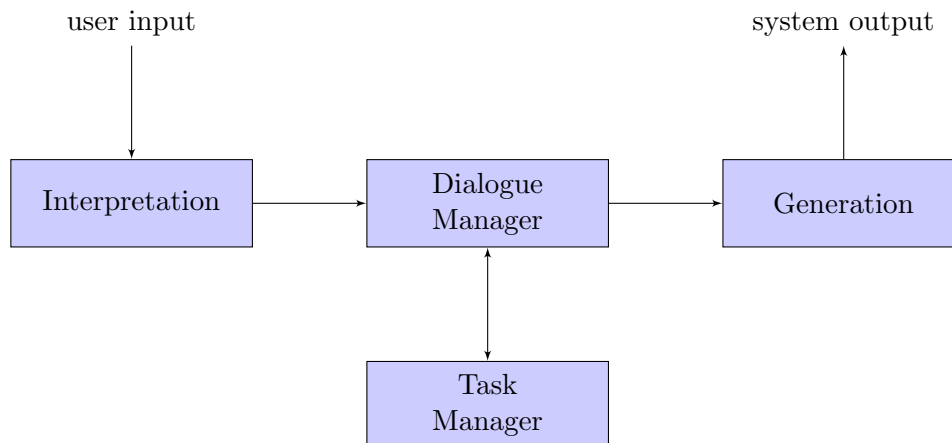


Figure 5.4: The prototype's system architecture

### 5.5.1 Interpretation

The main responsibility of the interpretation module is to look at the user's input and find out what it is that the user is doing. Is the user asking a question, making an assertion, greeting, thanking or giving a command? In other words, which speech act is the user performing?

In the ALICE chatbot the basic unit of knowledge is a category which directly maps user input to system output. In the chatbot/dialogue system hybrid such a direct mapping isn't possible. Instead of directly mapping input to output, the prototype, uses three different types of units of knowledge. The first type of unit is used by the interpretation module and is therefor appropriately named *interpret*.

#### Assertions

Consider the following example:

```
interpret ([
  pattern ([ star (A) , is , star (B) , ' . ' , star (R) ] ) ,
  meaning ([ you , tell , me , that , A , is , B ] ) ,
  srai (R)
] ) .
```

This unit matches assertions such as *"The boy is running"*, *"The price is 1234 dollars"*, and *"The sky is blue"*

Basically, any sentence which contains the word 'is' is recognized as an assertion. Everytime, the **pattern** part of an interpret unit matches user input the **meaning** part is added to the *THAT* field in the information state, awaiting further processing by the dialogue manager. So when the user inputs the phrase *the price is 1234 dollars* the contents that is added to the *THAT* field equals [you, tell ,me,that,the,price , is,1234, dollars ].

If the user adds more than one sentence at a time, the **srai** part tells the interpretation module to keep interpreting whatever comes after the first sentence.

Pronoun swapping takes place in the interpret module. Because the prolog implementation of ALICE doesn't support something similar to the <person> tag, pronoun swapping has to be done manually.

```
interpret ([
    pattern ([my, star (A) , is , star (B) , ' . ' , star (R)]) ,
    meaning ([you , tell , me , that , your , A , is , B]) ,
    srai (R)
]).
```

This unit matches assertions such as "*My name is Pedro*" and "*My father is an astronaut*". The input contains the word *my* which would refer to something that belongs to the user. All sentences that are stored in the *BEL*, *COM*, *THAT* or other parts of the information state must be interpreted as said from the perspective of the system. Thus the user's input "*My name is Pedro*" will be stored as "*Your name is Pedro*".

## Questions

Besides making assertions, a user may also ask questions.

```
interpret ([
    pattern ([ what , is , star (A) , ? , star (R)]) ,
    meaning ([ you , ask , me , what , A , is ]) ,
    srai (R)
]).
```

This unit would match questions like: "*What is the price?*", "*What is the meaning of life?*", and "*What is the color of the sky?*" If the user asks the question "*What is the price?*", this would result in the adding of [you,ask,me,what,the,price,is] to the *THAT* field.

## Answers to yes-no questions

All assertions are added to the *THAT* field as content starting with [you, tell ,me,that ,...]. All questions are added to this field as content starting with [you,ask,me ,...]. This way the dialogue manager can recognize the

speech act that the user has performed by looking at the start of each element in the *THAT* list and use this to decide how to handle this content.

Note that elements are added to the *THAT* field not only when interpreting input but also when the system generates output. Let's look at some more elaborate examples of interpret units.

```
interpret ([
  pattern ([yes, ' ', star(R)]),
  that ([i, ask, you, if, star(A)]),
  meaning ([you, tell, me, that, A]),
  srai(R)
]).
```

The interpret unit above includes a **that** part that contains a pattern that must match with the first element in the *THAT* field. This unit deals with situations when a user says "Yes". It makes sense that a user would say yes in response to a question asked by the system. The **that** part contains the pattern that matches with the question asked by the system.

To illustrate this, suppose that the system has just asked the question "*Do you like spaghetti?*". As discussed in more detail in section 5.5.2, the first element in the *THAT* field, after the system has asked this question will be [i,ask,you,if,you,like,spaghetti]. Now suppose that the user answers "Yes.", then the **pattern** part matches the user input and the **that** part matches the first element in the *THAT* field. This causes the content of [you,tell,me,that,you,like,spaghetti] to be added to the *THAT* field. This shows that an answer to a question is modelled as an assertion. This is different than how GoDiS deals with answers, where an answer is modelled as answer(yes).

In the same way the prototype deals with a "No" answer.

```
interpret ([
  pattern ([no, ' ', star(R)]),
  that ([i, ask, you, if, you, star(A)]),
  meaning ([you, tell, me, that, you, do, not, A]),
  srai(R)
]).
```

After asking the question "*Do you like spaghetti?*", the unit above would deal with the answer "No" by adding the content [you,tell,me,that,you,do,not,like,spaghetti] to the *THAT* field. In a similar way all answers to questions are modelled as assertions.

### Answers to wh-questions

A question such as "*What city do you want to go to?*" suggests that the answer to this question should be the name of a city. Similarly, the question

"What month is the first month of a year?" would expect as an answer the name of a month.

```
interpret ([
  pattern ([ star (NP) , ' . ' , star (R) ]) ,
  that ([ i , ask , you , what , star (T) , star (A) ]) ,
  believe ([ star (NP) , is , a , star (T) ]) ,
  meaning ([ you , tell , me , that , A , NP ]) ,
  srai (R)
]).
```

After the system asks the user the question "*What city do you want to go to?*", the first element in the *THAT* field will be [i,ask,you,what,city,you,want,to,go,to]. The unit above contains a **believe** part which checks if the *BEL* field in the information state contains a proposition that matches the pattern defined in **believe**. Suppose that the user answered "*London*", then the **believe** part only matches if there is a proposition in the *BEL* field whose content equals [london,is,a,city]. If it does, the content [you,tell,me,that,you,want,to,go,to,london] is added to the *THAT* field.

### Using DCG rules in patterns

Because the prototype uses predicates from the prolog ALICE implementation for pattern matching, we can use DCG grammar rules in units of the prototype as well. Assume we have defined grammar rules for parsing prepositions.

```
pre(place) —> [ at ] ; [ in ] ; [ on ] .
pre(time) —> [ before ] ; [ after ] ; [ at ] ; [ for ] ; [ since ] ; [ in ] ; [ on ] .
```

We can now use these from interpret units.

```
interpret ([
  pattern ([ syntax (pre , Pre , [ _ ] ) , star (B) , ' . ' , star (R) ]) ,
  that ([ i , ask , you , syntax (pre , Pre , [ _ ] ) , what , star (T) , star (A) ]) ,
  believe ([ star (B) , is , a , star (T) ]) ,
  meaning ([ you , tell , me , that , A , Pre , B ]) ,
  srai (R)
]).
```

This unit allows a user to respond to the question "*In what month do you want to leave?*" with "*In october*", or to the question "*At what time do the students have to be in class?*" with the answer "*At two o'clock*".

The advantage of including syntax rules in patterns is that it prevents us from having to include a separate interpret unit for every possible preposition.



## Ellipsis

In linguistics, the term **ellipsis** refers to the omission of one or more words from a sentence. Consider the following examples:

- John has a cat, and Sue [has] a dog.
- I am working on Monday, but [I am] not [working] on Tuesday.
- What is his name?  
I don't know [what his name is].
- I am dropping out of college.  
Why [are you dropping out of college]?

The text between brackets is optional and can be excluded. It is up to the interpret units to undo these omissions using interpret units. The previous sections showed some examples of this. For example, the previous unit interpreted the elliptical phrase "*In october*" as the proposition [you, want, to, leave, in, october].

```
interpret ([
  pattern ([why, ?, star (R)]) ,
  that ([i, tell, you, that, star (A)]) ,
  meaning ([you, ask, me, why, A, ]) ,
  srai (R)
]).
```

Suppose the system just told the user that the sky is blue. The first element in the *THAT* field is [i, tell, you, that, the, sky, is, blue]. This interpret unit will allow the user to ask the follow-up question *why?* The content that is added to the *THAT* field is [you, ask, me, why, the, sky, is, blue].

## Sanitizing input

The examples so far show that the interpretation module uses interpret units to find out the speech act that the user wants to perform (make an assertion, ask a question) and mold the user input into some sort of standard format and add it to the *THAT* field.

The interpret units above expect user input to be grammatical and free of spelling mistakes. It is to be expected that in practice user input sometimes won't be very grammatical at all. It may also contain spelling errors and profanity that one would like to filter out. An additional task of the interpretation module is to sanitize user input.

The following rule expands the contraction *would've* into *would have*.

```
interpret ([
  pattern ([ star (A) , would , '\ ' , ve , star (B) ]) ,
  srai ([A, would , have , B])
]).
```

In a similar way we can correct mistakes such as the common misuse of *your* instead of *you're* when *you are* is meant.

```
interpret ([
  pattern ([ star (A) , your , a , star (B) ]) ,
  srai ([A, you , are , a , B])
]).
```

This is similar to the AIML category at the end of section 2.5.2.

### 5.5.2 Generation

The generation module takes care of producing system output. Because the prototype already works with natural language internally there is often very little needed to produce grammatically correct natural language.

In the previous section, examples of interpret units were given. This section shows examples of generate units, another kind of unit of knowledge.

#### Assertions

```
generate ([
  agenda ( tell , [ star (A) ]) ,
  template ([A, '. ']) ,
  meaning ([ i , tell , you , that , A])
]).
```

This unit defines what should happen when the system plans to make an assertion. When the system is about to tell the user that for example *"The price is 1234 dollars"*, the first element on the *AGENDA* field in the information state will be `respond([the,price,is,1234,dollars])`. The `tell` in the **agenda** part indicates that this generate unit is used for assertions. The `star(A)` is the pattern that would in this case match with `[the,price,is,1234,dollars]`. The **template** part tells the system how to output the matched assertion. In this case the assertion is output as is and is appended by a period to mark the end of the sentence. The **meaning** part has the function as in the interpret units. It defines what should be added to the *THAT* field.

#### Questions

A more interesting example of a generate unit is the following one.

```
generate ([
```

```

agenda(ask,[how,i,can,star(T)]),
template(['How_can_I',T,?]),
meaning([i,ask,you,how,i,can,T])
]).

```

This unit matches when the prototype is about to ask a how-question. When this is the case, the *AGENDA* field's first element equals a raise or a findout. For example, the first element might be findout([how,i,can,jump,start,a,car]). Note that the system models issues as non-inverted questions. In standard English, questions are often formed by inverting the order of the subject and the verb. Because the prototype uses non-inverted questions to model issues, the subject and verb need to be inverted when outputting these questions.

Another example of a generate unit that inverts the subject and verb in questions is

```

generate([
  agenda(ask,[what,star([Obj]),you,are,star(T)]),
  template(['What',Obj,are,you,T]),
  meaning([i,ask,you,what,Obj,you,are,T])
]).

```

which would correctly deal with *AGENDA* elements such as findout([what,country,you,are,from]) or raise([what,color,you,are,wearing]).

To prevent us from having to add generate units for every subject/verb combination there is, we can also make use of DCG grammar rules like we did earlier. Also, sometimes before inverting a question we have to add the auxiliar *do*.

```

generate([
  agenda(ask,[what,star([Obj]),you,
syntax(verb,[Verb],[second,singular,present,indicative,Inf]),
star(T)]),
  think(not(member(Inf,[be,do,will,can,may]))),
  template(['What',Obj,'do_you',Verb,T,?]),
  meaning([i,ask,you,what,Obj,you,Verb,T])
]).

```

This unit would come into play when the first element of *AGENDA* would contain for instance findout([what,month,you,want,to,leave]). The pattern in the **agenda** part uses a DCG rule to match a verb. In this case the matched verb would be *want*. The **think** part is a backdoor to prolog and is used to ensure that the matched verb isn't an auxiliar because in that case there would be no need to add the *do* auxiliar. Another generate unit would match this scenario instead.

## Grounding

Generate units will also be used for grounding utterances.

```
generate ([
  agenda (icm : acc*pos , []) ,
  template ( [ 'Okay. ' ] ) ,
  meaning ( [ i , tell , you , that , i , can , process , what , you , say ] )
] ) .
```

The system might have an ICM action on its *AGENDA* to inform the user that it accepts the user input. The unit above would output *Okay* and add some descriptive content on what it just did to the *THAT* field.

When the system intends to give some implicit understanding feedback the following unit will help with that.

```
generate ([
  agenda (icm : und*pos , Q) ,
  template ( [ Q , ' . ' ] ) ,
  meaning ( [ i , tell , you , that , i , understand , pragmatically , that |Q| ] )
] ) .
```

This unit would be used when the first *AGENDA* element matches for instance icm:und\*pos:[you,had,business,class,in,mind]. The generation module would output "*you had business class in mind*" and append *THAT* with [i , tell , you , that , i , understand , pragmatically , that , you , had , business , class , in , mind].

### 5.5.3 Dialogue Manager

This section describes the dialogue manager module of the prototype. Because the dialogue manager is very similar to the one in GoDiS which has been described in detail in section 3.5.2, this description will only focus on the differences.

The changes in information state have already been detailed in section 5.4 and won't be repeated here.

#### Dialogue moves

The sections interpretation and generation gave some examples of dialogue moves. Two examples were the act of the user asserting something, and the system asking a question. These dialogue moves performed by either the user or the system always end up in the *THAT* field in the information state. An assertion by the user is represented as [you,tell,me,...] and the act of the system asking a question as [i,ask,you,...]. In a similar way all dialogue moves, including ICM moves have a textual representation.

Some additional examples are:

- The system greeting the user:  
[i,greet,you]
- The system asking for interrogative feedback (an ICM move):  
[i,ask,you,if ,...]
- The system informing the user that it is loading a plan:  
[i, tell ,you,that,i,load,a,plan]

The advantage of a textual representation of dialogue moves and the fact that they end up in the *THAT* field allows them to be pattern matched by including a **that** part in interpret units.

Note that performing the ICM move of asking for interrogative feedback has the same textual representation as the dialogue move of asking a yes-no question.

Another thing to keep in mind is that the textual representation of the dialogue move doesn't have to be similar to the output that is shown to the user. For instance, the loading of a plan, could output something like "*Let's see...*", while the representation that is added to the *THAT* field equals that of the last example above.

## Issue modelling

As stated in section 5.5.2 the system models issues as non-inverted questions. The reason for doing this is that the part after [you,ask,me, in [you,ask,me, what,the,price,is] is also a non-inverted questions. By choosing this format the system can easily isolate the issue from the dialogue move.

The different kinds of questions and corresponding issues are:

- Yes-no question asked by the user such as "*Do I need a visa?*" and "*Is the sun already up?*" are modelled as [if,you,need,a,visa] and [if,the,sun,is,already,up].
- Wh-questions are questions that often start with an interrogative pronoun, such as *who*, *what*, *when*, *where*, *why*, *which*, or *how* and which cannot be answered by a simple *yes* or *no*. Examples are: "*What is your name?*" and "*Where is the hotel?*" These can be modelled as [what,your,name,is] and [where,the,hotel,is]. Wh-questions don't always start with a Wh-word. They might also begin with a preposition such as in the question "*In what year did Obama become president?*" which could be represented on the *QUD* as [in,what,year,obama,became,president]. This example also shows that questions sometimes specify what answer is expected. In this case the answer would be the year 2009.

- Questions which offer a fixed choice of alternatives. Examples of this are *"Does the child want to go swimming or does the child want to sleep?"* and *"Is that man a janitor or a professor?"* The non-inverted question forms would be [ if , the , child , wants , to , go , swimming , or , if , the , child , wants , to , sleep ] and [ if , that , man , is , a , janitor , or , if , that , man , is , a , professor ]. Observe how a sentence like *"Is that man a janitor or a professor?"* is short for *"Is that man a janitor or is that man a professor?"*. As described previously, interpret units have the task of recognizing and expanding these and other forms of ellipsis.

## Resources

The example domain that GoDiS implements is that of a travel agent that can answer questions about ticket prices and visa requirements. To add a new domain, system designers would have to define one or more plans, a lexicon and a resolves relation. See section 3.5.2 for details. This section describes how to define a domain that works with the hybrid prototype.

The equivalents of the plans for ticket prices and visa requirements that work with the prototype look like this:

```
plan ( [ what , the , price , is ] ,
      [
        findout ( [ how , you , want , to , travel ] ) ,
        findout ( [ what , city , you , want , to , go , to ] ) ,
        findout ( [ what , city , you , want , to , go , from ] ) ,
        findout ( [ in , what , month , you , want , to , leave ] ) ,
        findout ( [ on , what , day , you , want , to , leave ] ) ,
        findout ( [ what , class , you , had , in , mind ] ) ,
        consultDB ( [ what , the , price , is ] )
      ] ) .

plan ( [ if , you , need , a , visa ] ,
      [
        findout ( [ what , city , you , want , to , go , to ] ) ,
        findout ( [ what , country , you , are , from ] ) ,
        consultDB ( [ if , you , need , a , visa ] )
      ] ) .
```

Because the issues in the plans above are represented as non-inverted questions, it is no longer necessary to define or extend the lexicon for every new domain that is added to the system. A default set of interpret and generate units covering the English language define how to turn an issue such as [ how , you , want , to , travel ] into the question *"How do you want to travel?"* and a question like *"Do I need a VISA?"* into the issue [ if , you , need , a , visa ].

When a user enters the question "*What is the price?*", this would end up in the *THAT* field as [you,ask,me,what,the,price,is]. After that the system moves the issue [what,the,price,is] onto the *QUD* field and loads the plan above.

The user input is recognized because of the following interpret unit.

```
interpret ([
  pattern ([ what , is , star (A) , ? , star (R) ]) ,
  meaning ([ you , ask , me , what , A , is ]) ,
  srai (R)
]).
```

Because there are many (often ungrammatical) ways in which a user may ask for ticket prices, it makes sense to define additional interpret units that reduce other ways of asking "*What is the price?*" to something that the system knows how to interpret. For instance the unit

```
interpret ([
  pattern ([ star ( _ ) , price , star ( _ ) , ' . ' , star (R) ]) ,
  srai ([ what , is , the , price , ? , R ])
]).
```

would reduce any input containing the word *price* to "*What is the price?*". So by defining the interpret unit above the system can from now on also interpret inputs such as "*Price info please*" and "*Gimme the ticket price*".

## Resolvedness

In GoDiS the dialogue system designer has to define a relation to determine if an answer resolves an open question. If the wh-question ?X.dest\_city(X) can be resolved by either a proposition such as dest\_city(paris) or an underspecified propositional content such as paris this would have to be defined by this relation.

In the prototype this relation takes the form of another unit of knowledge called resolve.

```
resolve ([
  query ([ how , star (A) ]) ,
  match ([ star (A) , by , star (B) ]) ,
  fact ([ A , by , B ])
]).
```

Let's assume that the system has asked the user how he or she wants to travel. The first issue on the *QUD* now equals [how,you,want,to,travel]. If the user answers that he or she wants to travel by plane then the proposition [you,want,to,travel,by,plane] will be added to the *BEL* field. Now the issue can be resolved using the resolve unit above. The **query** part matches the [how,you,want,to,travel] and the **match** part will match with

the proposition [you,want,to,travel,by,plane]. Because both parts match the question is considered resolved and the issue can be removed from the *QUD* field.

Some examples of issues and resolving propositions this unit matches are:

- Issue 1: [how,you,want,to,repair,your,bike]  
 Proposition 1: [you,want,to,repair,your,bike,by,patching,your,flat,tire]
- Issue 2: [how,you,got,so,skinny]  
 Proposition 2: [you,got,so,skinny,by,dieting]
- Issue 3: [how,you,can,afford,this,house]  
 Proposition 3: [you,can,afford,this,house,by,working,day,and,night]

When writing resolve units, it makes sense to think of wh-words as placeholders for a phrase. In the examples above *how* is a placeholder for a prepositional phrase. In the sentences "*Who's your daddy?*" and "*What is that man riding?*", the words *who* and *what* would be placeholders for noun phrases such as *He* and *a horse*.

Suppose that the issue to resolve is [how,you,want,to,travel] and the *BEL* doesn't contain the proposition [you,want,to,travel,by,plane] but it does contain [you,would,like,to,travel,by,plane]. Because *want* and *would like* mean the same thing it is useful to also define the following resolve unit.

```
resolve ([
  query ([how,you,want,star(A)]),
  match ([you,would,like,star(A),by,star(B)]),
  fact ([you,want,A,by,B])
]).
```

This example also clarifies the **fact** part that paraphrases the proposition in *BEL* to a resolving answer.

The following two units would resolve questions which offer a fixed choice between two alternatives. An example of such a question would be: "*Do you want to go out with friends or do you want to stay at home with me?*".

```
resolve ([
  query ([if,star(A),or,if,star(-)]),
  match ([star(A)]),
  fact ([A])
]).
```

```
resolve ([
  query ([if,star(-),or,if,star(A)]),
  match ([star(A)]),
  fact ([A])
]).
```



])).

The first of these two units would resolve this issue if the *BEL* field contains a proposition equal to [you,want,to,go,out,with,friends], that is, the first option, while the second unit would resolve the issue if the *BEL* field holds a proposition that equals [you,want,to,stay,at,home,with,me].

## Grounding

In GoDiS, update rules such as the `integrateUsrAsk` rule use a score variable to determine if it should give grounding feedback and if this feedback should be implicit or explicit. This score is determined by the speech recognizer based on how well spoken user input could be discerned.

Very few chatbots however accept spoken input and get their input from what a user types into a chat window either on a webpage or in an instant messaging program. For now, the prototype just deals with typed user input and uses a different approach to calculate a score. The following rule calculates a score based on how many words it had to skip before it found a sequence of words that would match with one of the interpret units.

```
rule nibble forward
  if
    gid( _ ) &
    input( [Word| Tail] ) &
    score( Score ) &
    length( Length ) &
    skipped( Skipped )
  then
    prolog( Skipped2 is Skipped + 1 ) &
    prolog( Score2 is 1 - ( Skipped2/Length ) ) &
    remove score( Score ) &
    remove skipped( Skipped ) &
    add score( Score2 ) &
    add skipped( Skipped2 ) &
    remove input( [Word| Tail] ) &
    add input( Tail ).
```

Notice that the rule above has a somewhat different syntax than the ones in GoDiS. This is due to the fact that the prototype uses MIKE as a rule engine and not TrindiKit.

When no interpret unit matched with the user input, this rule removes the first word and retries to find a match with one of the many interpret units. Everytime a word is taken away, the **score** goes down a little and the variable **skipped** is increased. The **score** variable is determined by dividing the number of skipped words by the initial length of **input**. The result of

this is subtracted from 1, leaving a score that can be used by rules such as `integrateUsrAsk`.

System: What city do you want to go to ?  
 User: ksd sdv we v wf sdv london  
 System: Do you want to go to london ?  
 User: yes  
 System: Okay.  
         What city do you want to go from ?  
 User: i want to go from paris  
 System: Okay.  
         in what month do you want to leave ?  
 User: sdv i want to leave in october  
 System: Okay.  
         you want to leave in october .

When the user answers "*ksd sdv we v wf sdv london*" the nibble rule removes words until the word *london* is left and is matched as an answer to the question "*What city do you want to go to?*". Because it encountered quite some noise before the actual answer, the **score** decreased a lot which caused the system to output the explicit grounding feedback "*Do you want to go to london?*". When the user answers "*I want to go from Paris*", no grounding feedback is produced because the input was instantly matched as a valid answer to the question "*What city do you want to go from?*" When the user answers "*sdv I want to leave in October*" only *sdv* is skipped which causes the system to produce implicit grounding feedback.

Everytime the system reads another string of user input, the variable **score** is reset to 1, **length** is set to the length of **input** and **skipped** is reset to 0.

A user might also produce input such as "*Uh, well,.. actually to London*". Because the use of filler words such as *uh*, and *well* are so common we should prevent the system from causing these words to lower the recognition score by adding the following interpret unit.

```
interpret ([
  pattern ([ star ([W]) , star (R) ] ) ,
  think (member (W, [ ' . ' , ' , ' , ' ? ' , ' ! ' , uh , um , er , well , like ,
                                     so , actually , basically , right ] ) )
  srai (R)
]).
```

This unit does the same as the nibble rule but only for filler words that are safe to skip without lowering the **score** variable.

Besides calculating a score for grounding purposes, the nibble rule, together with the interpret unit above, also make the system more robust against superfluous input.

## Question accommodation

The three main types of question accommodation are local accommodation, global accommodation and dependent accommodation. The system for now only support the first two types. Besides these three types we can also discern a fourth type which is question re-accommodation.

Local accommodation is the resolving of elliptical answers. This is taken care of by using appropriate interpret units. Consider the following:

```
interpret ([
    pattern ([ star (NP) , star (R) ]) ,
    that ([ i , ask , you , what , star (T) , star (A) ]) ,
    believe ([ star (NP) , is , a , star (T) ]) ,
    meaning ([ you , tell , me , that , A , NP ]) ,
    srai (R)
]) .
```

The unit above allows the user to answer "*London*" to the question "*What city do you want to go to?*".

Global accommodation refers to the possibility to answer questions before they're raised. This is taken care of automatically. Every assertion the user makes is added to the *BEL* field. Whenever an issue is added to the *QUD* the system tries to resolve this by inspecting the *BEL* field first. Only when no resolving fact is found the system explicitly raises the issue.

Question re-accommodation refers to the possibility of revising previously given information. To detect conflicting information in the *BEL* field the system uses clauses that specify patterns of conflicting facts.

```
conflict ([ [ there , is , one , star (Adj) , star (Singular) ] ,
            [ there , are , syntax(ordnum , - , [ - ] ) , star (Adj) ,
              star (Plural) ] ]) :-
    singular_plural (Singular , Plural) .
conflict ([ [ there , are , syntax(ordnum , - , [ A ] ) , star (N) ] ,
            [ there , are , syntax(ordnum , - , [ B ] ) , star (N) ] ])
:-
    A \= B .
conflict ([ [ star (A) , not , star (B) ] , [ star (A) , star (B) ] ]) .
```

The first clause would detect conflicting propositions like [there, is, one, cold, beer] and [there, are, two, cold, beers]. The second would detect conflicting propositions such as [there, are, two, cold, beers] and [there, are, three, cold, beers]. The third detects conflicts like [i, am, not, 20, years, old] and [i, am, 20, years, old].

The conflict clauses are used by the following rules:

```
rule retract forward
    if
```

```

gid(-) &
qud([- ,Q|-]) &
is_empty(agenda) &
prolog(lexsem(tell(user , self ,A) ,T)) &
fst(that ,T) &
prolog(conflict(C)) &
prolog(permutation(C,[PA,PNotA])) &
pattern(A,PA) &
in(bel ,NotA) &
pattern(NotA,PNotA) &
prolog(plan(Pattern ,Plan)) &
prolog(pattern(Q,Pattern))
then
push(agenda ,icm:acc*pos) &
pop(qud) &
set(plan ,Plan) &
del(bel ,NotA) &
add(bel ,A) .

rule retract forward
if
gid(-) &
prolog(lexsem(tell(user , self ,A) ,T)) &
fst(that ,T) &
prolog(conflict(C)) &
prolog(permutation(C,[PA,PNotA])) &
pattern(A,PA) &
in(bel ,NotA) &
pattern(NotA,PNotA)
then
push(agenda ,icm:acc*pos) &
del(bel ,NotA) &
add(bel ,A) .

```

The first rule executes when the user revises some information and a plan is loaded. If a plan is loaded the dialogue manager has to reload the plan, because the revised information may alter the outcome of the plan. The second rule executes when the user enters information but no plan is active. In this case no plan has to be reloaded and only updates to the *BEL* field take place.

## 5.6 Implementation of the case study

This section describes how the case study is implemented. A full listing of the statistics domain implementation can be found in appendix A.

### 5.6.1 Decision tree

The prototype can assist the user in deciding which statistical test to use. This is done by following a decision tree where end points correspond to specific statistical techniques. The decision tree that the prototype uses comes from an introductory statistics book[78] and is replicated in figure 5.5.

### 5.6.2 Plans

The following plan is loaded as soon as the system starts and makes sure the system tries to find out how it can assist the user.

```
plan (top ,
[
  raise ([how , i , can , help , you] ) ,
  findout ([ if , you , want , to , know , which , statistical ,
    technique , you , should , use ] )
]) .
```

The findout action in this plan could be extended to something like findout([ if , you , want , to , know , which , statistical , technique , you , should , use , or , if , you , want , to , know , how , you , can , check , for , normality ]) to make the system output the options the system could be used for. More options could be added. This would however make the system somewhat menu-based where the user is supposed to pick one of the available options instead of feeling free to voice his or her own questions.

The plan that is used to determine which test is appropriate is based on the decision tree as depicted in figure 5.5. Translating such a decision tree into a plan is straightforward. The following code fragment only shows the highlights of this plan.

```
plan ([ which , statistical , test , you , should , use ] ,
[
  findout ([ how , many , dependent , variables , there , are ] ) ,
  ifthenelse ([ there , is , one , dependent , variable ] ,
  [
    findout ([ if , the , dependent , variable , is , continuous ,
      or , if , the , dependent , variable , is , categorical ] ) ,
    ifthenelse ([ the , dependent , variable , is , continuous ] ,
    [
      findout ([ how , many , independent , variables , there ,
        are ] ) ,
```

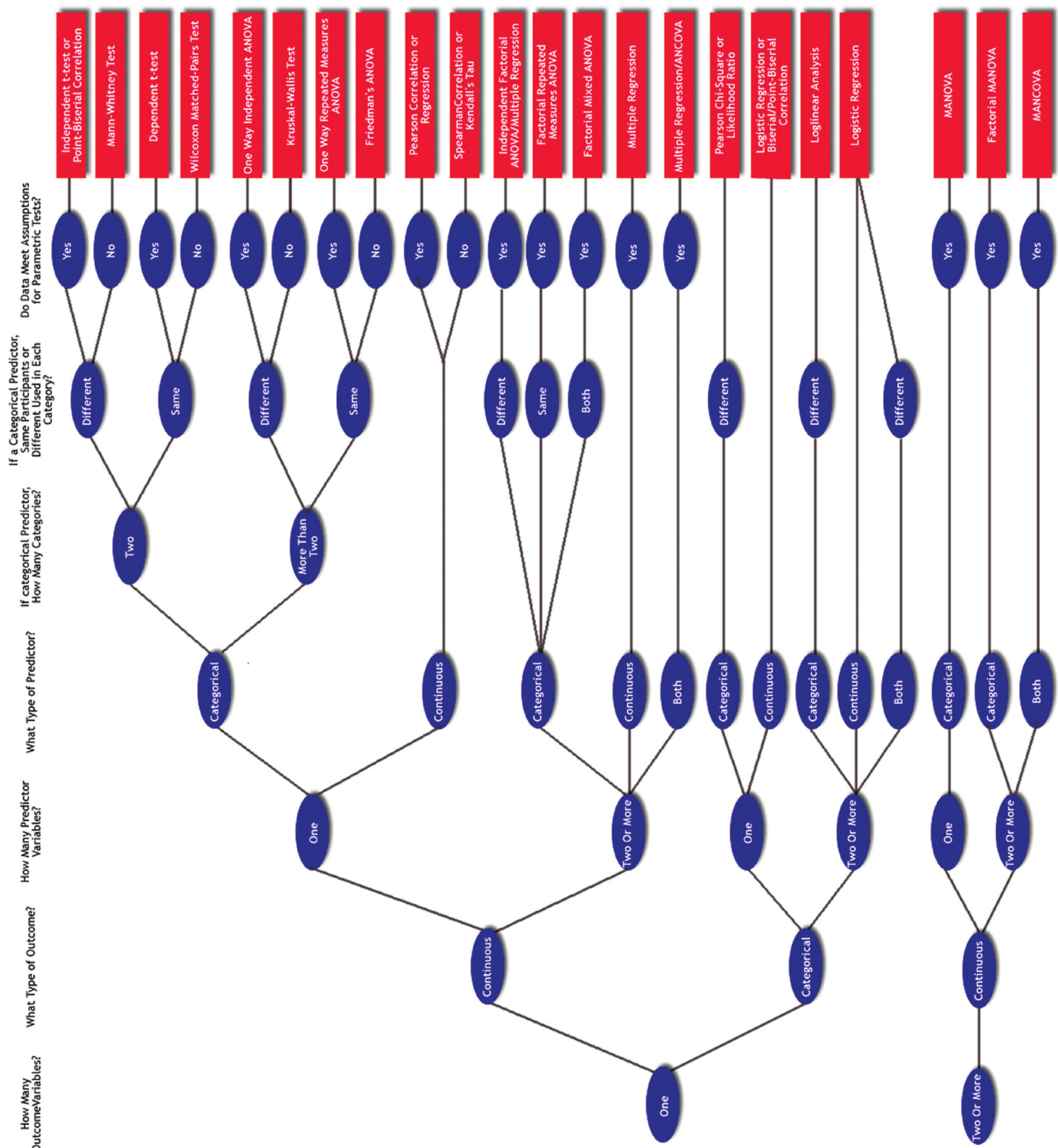


Figure 5.5: Decision tree

```

        ifthenelse ([there , is , one , independent , variable] ,
        [
...
            ifthenelse ([the , data , meets , assumptions ,
                for , parametric , tests] ,
            [add ([you , should , use , the , independent , t ,
                test]) ] ,
            [add ([you , should , use , the , mann , whitney ,
                test]) ])
        ] ,
...
    ])
    ])
    ])
    ]) .

```

A plan for getting advice on the best way to determine normality of data was also added.

```

plan ([how , you , can , check , for , normality] ,
[
    findout ([ if , you , want , to , check , for , normality , visually
        , or , if , you , want , to , test , for , normality]) ,
    ifthenelse ([you , want , to , check , for , normality , visually
        ] ,
    [ add ([you , can , check , for , normality , by , inspecting , the
        , histogram , or ,
            you , can , check , for , normality , by , inspecting ,
            the , p , p , plot]) ] ,
    [ findout ([ if , you , have , more , than , 50 , cases]) ,
        ifthenelse ([you , have , more , than , 50 , cases] ,
        [add ([you , can , test , for , normality , by , using , the ,
            kolmogorov , smirnov , test]) ] ,
        [add ([you , can , test , for , normality , by , using , the ,
            shapiro , wilk , test]) ])
    ])
    ]) .

```

### 5.6.3 Additional units

The system already contains a good set of interpret, resolve and generate units that cover a reasonable part of the English language. The sections below discuss some of the domain specific units that needed to be added to have the system converse in the statistics domain.

## Interpret units

The following interpret units are used to make sure the system understands a wide variety of user input.

```
interpret ([
    pattern ([ star (-) , test , star (-) , star (R) ]) ,
    that ([ i , ask , you , how , i , can , help , you ]) ,
    srai ([ which , statistical , technique , should , i , use , ? , R ])
]).
```

This interpret unit matches any input that contains the word test, such as "*What test can i use?*", "*Which is a good test?*", and "*Help me to pick a good statistical test*". There are many ways a person can ask the system for advice on a statistical technique, and this interpret pattern will match many of them. The srai rule will rewrite this input to "Which statistical technique should I use?".

Some other interpret units that were added to the domain were:

```
interpret ([
    pattern ([ star (-) , what , star (-) , dependent , star (-) ]) ,
    srai ([ what , is , a , dependent , variable , ? ])
]).
```

```
interpret ([
    pattern ([ star (-) , what , star (-) , continuous , star (-) ]) ,
    srai ([ what , is , a , continuous , variable , ? ])
]).
```

```
interpret ([
    pattern ([ star (-) , what , star (-) , categorical , star (-) ]) ,
    srai ([ what , is , a , categorical , variable , ? ])
]).
```

These units match user input such as "*What does continuous mean?*", and "*What is meant by dependent variable?*" and reduce it to a question the system can answer.

To make sure the system interprets questions about normality correctly the following interpret units were also added.

```
interpret ([
    pattern ([ star (-) , how , star (-) , normality , star (R) ]) ,
    meaning ([ you , ask , me , how , you , can , check , for ,
        normality ]) ,
    srai (R)
]).
```



```
interpret ([
    pattern ([ star ( _ ) , how , star ( _ ) , normally , star (R) ]) ,
    meaning ([ you , ask , me , how , you , can , check , for ,
              normality ]) ,
    srai (R)
]) .
```

These units match user input like *"How to check for normality."* and *"How can I know if data is normally distributed?"*.

### Resolve units

A fact such as [you,can,check,for,normality,by,inspecting,the,histogram] resolves the issue [how,you,can,check,for,normality]. This is covered by one of the default resolve units. To have a fact such as [you,can,test,for,normality,by,using,the,shapiro,wilk,test] resolve the issue above the following additional resolve unit is needed.

```
resolve ([
    query ([ how , you , can , check , for , normality ]) ,
    match ([ you , can , test , for , normality , by , using , star (A)
            ]) ,
    fact ([ you , can , test , for , normality , by , using , A ])
]) .
```

### Generate units

A lot of output is already covered by the default generate units. A system greet is taken care of by the following generate unit.

```
generate ([
    agenda (greet , []) ,
    template ([ 'Hello . ' ]) ,
    meaning ([ i , greet , you ])
]) .
```

The system could be made more proactive by changing the unit above and have it immediately output its purpose. The unit above could be changed to something like this:

```
generate ([
    agenda (greet , []) ,
    template ([ 'Welcome _ to _ the _ statistics _ tutor . ' ])
    ,
    meaning ([ i , greet , you ])
]) .
```

It was decided not to do this in order to emphasize that the system isn't limited to the closed domain of statistics but can respond to other input as well.

#### 5.6.4 Facts

Things the system believes are loaded on startup into the *BEL* field. For the current domain implementation, many facts like the following were defined.

```
bel([point,biserial,correlation,is,a,statistical,
    technique])).
bel([loglinear,analysis,is,a,statistical,technique])).
bel([manova,is,a,statistical,technique])).
bel([factorial,manova,is,a,statistical,technique])).
bel([mancova,is,a,statistical,technique])).
bel([a,categorical,variable,is,a,variable,that,can,
    take,on,a,limited,number,of,values])).
bel([a,continuous,variable,is,a,variable,that,can,have
    ,an,infinite,number,of,values,between,two,points])).
bel([a,dichotomous,variable,is,a,variable,which,has,
    only,two,categories])).
bel([a,dependent,variable,is,the,outcome,or,effect,
    variable])).
bel([an,independent,variable,is,the,variable,that,is,
    changed,to,test,the,effect,on,a,dependent,variable
    ])).
```

### 5.7 Implementation of the demo interaction

The demo interaction makes use of the statistics domain described above, but also shows how the system can interface with a webservice. A webservice for getting times around the world was used. The code needed for answering questions about different times around the world and calling the web API can be found in appendices B and C.

This code also gives an example of how to call prolog DCG grammar rules from interpret units and plans.

## Chapter 6

# Testing of the system

This chapter describes the interactions between the participants and the system. Three participants conversed with the system. Each interaction is shown and has been annotated to describe what's going on inside the program. After each transcript the interpret, resolve and generate units and plans are described that have been added to the system.

The chapter ends with an interaction that demonstrates some of the dialogue phenomena the system supports.

### 6.1 Participant #1

The first participant was Mathilde, a twenty year old sophomore student of Archeology. Last year she took an introductory course in statistics and understands basic concepts such as correlation and hypothesis testing.

#### 6.1.1 Transcript

**System** Hello.  
How can I help you ?

*When the program starts the agenda contains the action greet which causes the system to greet the user as soon as it starts. Furthermore, a default plan is loaded that has as its first action to raise the question 'How can I help you?'*

**Mathilde** I want to test the correlation between sex and the rate of dropout for a course. How can I do that?

*Although Mathilde tries her best to explain the goals of the analysis, the system only matches the word "test", and reduces the input to "Which statistical technique should I use?". After that another interpret unit matches this user input and adds [you,ask,me,which, statistical , technique ,you,should, use] to the THAT field. The system pushes [which, statistical , technique ,you,should,use] onto the QUD.*

**System** Let's see.

Okay.

How many dependent variables are there ?

*Because the system has no direct answer to this question, a plan is loaded to resolve this issue. The loading of a plan has the effect of the system outputting "Let's see.". The first item in this plan is findout ([how,many,dependent,variables,there,are]) so the system starts by asking the user for the number of dependent variables. When the system outputs "How many dependent variables are there?", it also adds [i,ask,you,how,many,dependent,variables,there,are] to the THAT field.*

**Mathilde** 1

*The system is able to deal with elliptical input like this, because the top item on the THAT stack now contains [i,ask,you,how,many,dependent,variables,there,are]. The system now adds [you,tell,me,that,there,is,one,dependent,variable] to the THAT field. The fact [there,is,one,dependent,variable] is subsequently added to the BEL set. The next item in the plan is an if/then/else construct. Because the if part ([there,is,one,dependent,variable]) matches a fact in the BEL set, the 'then' part of the construct is now executed.*

**System** Okay.

Is the dependent variable continuous or categorical ?

*The plan continues executing and outputs another question for the user to answer. Every answer results in information being added to the BEL field. These added facts determine the flow of the plan execution.*

**Mathilde** Categorical

**System** Okay.

How many independent variables are there ?

**Mathilde** 1

**System** Okay.

Is the independent variable continuous or categorical ?

**Mathilde** Categorical

**System** Okay.

is the experimental design an independent measures design ?

**Mathilde** What is an independent measures design?

**System** Okay.

I don't know what an independent measures design is .

is the experimental design an independent measures design ?

*The system has no knowledge of what an independent measures design is. This is kind of silly, so this fact will be added to the system's knowledge base.*

**Mathilde** Yes.

*After Mathilde answers the last question, the plan reaches an end point, and a fact is added to the BEL field indicating which statistical technique(s) to use.*

**System** Okay.

You should use pearson chi square or likelihood ratio .

*Because there is a fact in the BEL field, which resolves an open issue on the QUD stack, the system decides to share this fact with the user. Then it removes the issue from the QUD field and copies the fact to the COM field indicating that the system now knows this is shared knowledge.*

**Mathilde** What is a pearson chi square?

**System** Okay.

I don't know what a pearson chi square is .

*The BEL set contains the fact [pearson,chi,square,is,a, statistical ,technique ] so it was expected that the system would answer "Pearson chi square is a statistical technique". This doesn't happen because Mathilde added an unnecessary article (a) in her question. If she would have asked "What is pearson chi square?", the system would have answered "Pearson chi square is a statistical technique.". This is something that can easily be fixed by adding a resolve unit as will be described under system additions below.*

**Mathilde** Do you have a girlfriend?

**System** Okay.

I don't know if i have a girlfriend .

*After the user asks "Do you have a girlfriend?", the issue [if,i,have,a, girlfriend ] becomes the top item on the QUD stack. No fact such as [i ,have,a, girlfriend ] or [i,do,not,have,a, girlfriend ] is found in the BEL field, so the system concludes it doesn't know. Then Mathilde does something interesting.*

**Mathilde** You have a girlfriend. Her name is Venna.

**System** Okay.

*Without having been instructed to do so, she provides the system with new information. Two facts are added to the BEL field, namely [i,have,a, girlfriend ] and [her,name,is,venna].*

**Mathilde** Do you have a girlfriend?

**System** Okay.

i have a girlfriend .

**Mathilde** What is her name?

**System** Okay.

her name is venna .

*The system is now able to answer these two questions.*

**Mathilde** How old is she?

**System** I don't understand.

*The user asks one more question, and instead of answering "I don't know" as would have been expected, the system outputs "I don't understand.". The reason this happens is because no interpret unit has been added yet that can recognize user input that matches a pattern such as "How \* is \* ?". If she would have left out the question mark the system would have recognized the sentence as being an assertion, since there is an interpret pattern that matches user input using the pattern "\*\* is \*".*

### 6.1.2 System additions

After Mathilde's conversation with the system several additions were made to improve the quality of future interactions.

#### Addition #1

The first thing that was added were the following two facts.

```
bel([an,independent,measures,design,is,a,research,design,
    that,uses,two,or,more,conditions,with,different,
    participants,in,each,condition]).
bel([a,repeated,measures,design,is,a,research,design,in,
    which,multiple,measurements,are,made,on,each,
    experimental,unit]).
```

With these two facts added, the system is now able to answer questions like: *"What is an independent measures design?"*.

#### Addition #2

The following resolve unit was added to help the system answer *what* questions.

```
resolve([
    query([what,syntax(art,-,[]),star(A)]),
    match([star(A),star(B)]),
    fact([A,B])
]).
```

By adding this resolve unit, the system is able to answer questions that contain a superfluous article as in *"What is a pearson chi square?"*. After asking such a question it is added to the *QUD* stack as [what,a,pearson,chi,square,is]. This is matched by the query part in the resolve unit shown above. The `syntax(art,-,[])` fragment represents an article such as *a* and

*an.* The match part matches a fact in *BEL* such as [pearson,chi,square, is,a, statistical ,technique]. If these two parts match, the system knows it has a fact in it's *BEL* field that resolves the open issue and outputs this to the user.

### Addition #3

Mathilde was able to add new facts to the *BEL* field of the system and query for that information later. The facts that were added were [i,have, a, girlfriend ] and [her,name,is,venna]. It can be seen that some pronoun rewriting took place on the user input "*You have a girlfriend*". The pronoun *You* was changed to *I* because we want everything the system knows to be facts from the perspective of the system. In the second sentence "*Her name is Venna*" no pronoun rewriting took place, because the pronoun *her* refers to another person than the system or user. The system was able to answer the query "*What is her name?*", because the issue [what,her,name,is] is able to be resolved by the fact [her,name,is,venna]. Although this worked, imagine what would've happened if the user had asked "*What is the name of your girlfriend?*" instead? There is no way the system can know that "*her name*" in the fact refers to "*the name of your girlfriend*". Furthermore, *her* may refer to *your girlfriend* now, but it won't forever. If the subject of the conversation changes to other females, such as a mother, sister, aunt or even pets, the *her* pronoun should refer to these agents, depending on the state of the interaction. For this reason it makes sense to not add facts to the *BEL* field that contain pronouns such as *her*, *his*, *she*, *he*, *they*, and so on. Facts can contain pronouns such as *I*, *you*, *me*, *mine*, and *your* because for now it's assumed that the user and system remain constant within one conversation and these pronouns will always refer to the same agent.

To deal with input that contain third person pronouns in a robust way the system has to perform some kind of pronoun resolution. One way of dealing with pronoun resolution in AIML is by setting global variables such as *he*, *she* whenever the agent the pronoun refers to changes. This is done by including `<set var="she"></set>` fragments in AIML.

In this case we won't use global variables, but we let the system figure out what a pronoun refers to by looking at what was said earlier by inspecting the contents of the *THAT* field. Consider the following:

```
interpret ([
    pattern ([ her , star (S) , is , star (O) , ' . ' , star (R) ] ) ,
    that ([ star ([ - , - , - ] ) , that , i , have , syntax ( art , - , [ ] ) ,
        syntax ( noun , N , [ singular ] ) ] ) ,
    meaning ([ you , tell , me , that , the , S , of , my , N , is , O ] ) ,
    srai (R)
]) :- noun_infl ( - , - , - , - , N , - , - , - ) .
```

Imagine that a user has just uttered the phrase *"You have a girlfriend."*. The first thing the system does is add [you, tell , me, that, i, have, a, girlfriend ] to the *THAT* stack. When the user now says *"Her name is Venna."*, the interpret pattern above recognizes the *her* pronoun and sees that previously something was said about a female agent noun. The meaning part of the interpret unit above assembles these two pieces of information into [the, name, of, your, girlfriend , is , venna]. This causes the system to add this as a fact to the *BEL* field instead of [her, name, is, venna].

The following interpret unit was added also to the system.

```
interpret ([
    pattern ([ what , is , her , star (A) , ? , star (R) ] ) ,
    that ([ star ([ - , - , - ] ) , that , i , have , syntax ( art , - , [ ] ) ,
        syntax ( noun , N , [ singular ] ) ] ) ,
    meaning ([ you , ask , me , what , the , A , of , my , N , is ] ) ,
    srai (R)
]) :- noun_infl( - , - , - , - , N , - , - , - ).
```

This makes sure that when the user asks *"What is her name?"*, the *her* pronoun is also resolved. This is again done by inspecting the contents of the *THAT* field which would in this case contain [i, tell , you, that, i, have , a, girlfriend ]. The resulting meaning that this interpret pattern yields is [you, ask, me, what, the, name, of, my, girlfriend, is ].

The lexicon of the system contains noun\_infl/8 facts for common nouns for which both masculine and feminine forms exist. These are used in the interpret units to check if a verb is of feminine form.

Note how the interpret unit above is also able to make sense of input such as *"You have an aunt. What is her name?"*. The first sentence would add [you, tell , me, that, i, have, an, aunt] to *THAT*. The second sentence would in this case cause [you, ask, me, what, the, name, of, my, aunt, is] to be added to the *THAT* stack.

It makes sense for the system to know whether or not it has a girlfriend so optionally we could add the following two lines of code to the system.

```
bel ([ i , have , a , girlfriend ] ) .
bel ([ the , name , of , my , girlfriend , is , venna ] ) .
```

#### Addition #4

The last addition that was made to the system involved enabling the system to recognize sentences such as *"How old is she?"*.

A first attempt would be:

```
interpret ([
    pattern ([ how , star (A) , is , star (B) , ? , star (R) ] ) ,
    meaning ([ you , ask , me , how , A , B , is ] ) ,
```



```

    srai(R)
  ] ).

```

The system is now able to answer a question like "*How old is the house?*" if the system would have the fact [the,house,is,23,years,old] in its *BEL* set. In this case the user asked "*How old is she?*". What this interpret unit would do in this case is push [you,ask,me,how,old,she,is] to the *THAT* stack. The system would only be able to resolve this issue if the *BEL* field would contain a fact like [she,is,22,years,old]. Because we established that the *BEL* set should not contain facts containing third person pronouns such as *he* and *she* we need to add additional interpret units that handle pronoun resolution.

```

interpret ([
  pattern ([how, star(A), is, she, ?, star(R)]),
  that ([star([_, -, -]), that, the, star(_), of, my,
        syntax(noun, N, [singular]), is, star(_)]),
  meaning ([you, ask, me, how, A, my, N, is]),
  srai(R)
]) :- noun_infl( -, -, -, -, N, -, -, -).

```

```

interpret ([
  pattern ([how, star(A), is, she, ?, star(R)]),
  that ([star([_, -, -]), that, i, have,
        syntax(art, -, []), syntax(noun, N, [singular])]),
  meaning ([you, ask, me, how, A, my, N, is]),
  srai(R)
]) :- noun_infl( -, -, -, -, N, -, -, -).

```

Both interpret pattern do pronoun resolution for the pronoun *she* as in "*How old is she?*". The first one matches when *THAT* contains something similar to [i, tell, you, that, the, name, of, my, girlfriend, is, venna]. The second one matches when *THAT* contains for example [i, tell, you, that, i, have, a, girlfriend].

Similar interpret units can be added that handle pronoun resolution in case input contains the pronoun *he*.

## 6.2 Participant #2

The second participant was Lennart, a 33 year old research technician at the department of Genetics of the University Medical Center Groningen. He is also a part-time student of Philosophy. Three weeks prior to interacting with the system Lennart attended an SPSS course.

### 6.2.1 Transcript

**System** Hello.  
How can I help you ?

**Lennart** How can I compare the dropout rate with the gender of the students?

**System** I don't understand.  
Do you want to know which statistical technique you should use ?

*Lennart asks a question that the system doesn't understand. The user input didn't contain a keyword like "test" from which the system could deduce the meaning of the input. The next action in the default plan however is to findout if the user wants to know which statistical technique he should use, so it asks this question.*

**Lennart** yes

*This resolves the issue of the system wanting to know how it can help the user.*

**System** Let's see.  
Okay.  
How many dependent variables are there ?

**Lennart** 1

**System** Okay.  
Is the dependent variable continuous or categorical ?

**Lennart** categorical

**System** Okay.  
How many independent variables are there ?

**Lennart** 1

**System** Okay.  
Is the independent variable continuous or categorical ?

**Lennart** categorical

**System** Okay.  
is the experimental design an independent measures design ?

**Lennart** what is an independent measures design?

*Lennart asks the same question as Mathilde did before. Because some facts have been added to it's BEL set the system is now able to answer this question.*

**System** Okay.  
an independent measures design is a research design that uses

two or more conditions with different participants in each condition .

is the experimental design an independent measures design ?

**Lennart** yes

**System** Okay.

You should use pearson chi square or likelihood ratio .

**Lennart** how can i compare the quality of two runs?

**System** I don't understand.

*The statistics question that Lennart asks is related to his work in genome sequencing. The system doesn't understand his question, which is basically about finding an appropriate statistical technique similar to the interaction he has been having until now. Ideally the system would again go into asking a series of questions about the nature of the data and come up with an appropriate technique. This could be done by adding a pattern which would match input containing a keyword like "compare". In this case however the statistical techniques suitable for his problem are quite advanced and not covered by the decision tree this program implements. The system answers "I don't understand." because there are no interpret units that match a sentence like this one.*

**Lennart** who wrote this program?

**System** I don't understand.

*The system doesn't contain an interpret unit that matches the input "Who wrote this program?".*

**Lennart** how old is this program?

**System** Okay.

I don't know how old this program is .

*This time the system understands the question, but has no knowledge or plan to answer it.*

## 6.2.2 System additions

### Addition #1

To make the system understand questions that start with "How can I..." the following interpret unit is added.

```
interpret ([
    pattern ([how, can, i, star(A), '?', star(R)]),
    meaning ([you, ask, me, how, you, can, A]),
    srai(R)
]).
```

In the interaction above the system would, after the user asked: *"How can I compare the quality of two runs?"*, answer *"I don't know how you can compare the quality of two runs."*

### Addition #2

To make the system understand the question *"Who wrote this program?"* another interpret unit was added.

```
interpret ([
    pattern ([who, star(A), '?', star(R)]),
    meaning ([you, ask, me, who, A]),
    srai(R)
]).
```

This unit also matches input such as *"Who built the pyramids?"*, and *"Who invented Velcro?"*.

For the system to be able to answer *"Who wrote this program?"*, the following fact also needs to be known.

```
bel ([aswin, wrote, this, program]).
```

### Addition #3

The system recognizes the question *"How old is this program?"*, but has no knowledge in its *BEL* field to answer is. This is solved by adding:

```
bel ([this, program, is, half, a, year, old]).
```

## 6.3 Participant #3

The third participant was Fleur. Fleur is 28 and in the second year of her PhD at the University Medical Center Groningen. She did her Master's in Biomedical Sciences and has a reasonable working knowledge of statistics.

### 6.3.1 Transcript

**System** Hello.

How can I help you ?

**Fleur** I want to investigate wheter there is a difference between dropout rates between genders. What kind of test should I use?

*Fleur tries to explain what she wants to do. All input but the word "test" is ignored. This word triggers the system to activate the plan that tries to find out which statistical technique could be used.*

**System** Let's see.  
Okay.  
How many dependent variables are there ?

**Fleur** 1

**System** Okay.  
Is the dependent variable continuous or categorical ?

**Fleur** caterogrial

*Fleur misspells "categorical" so the system doesn't understand her. To deal with common misspellings an interpret unit will be added.*

**System** I don't understand.  
Is the dependent variable continuous or categorical ?

**Fleur** categorical

**System** Okay.  
How many independent variables are there ?

**Fleur** 1

**System** Okay.  
Is the independent variable continuous or categorical ?

**Fleur** categorical

**System** Okay.  
is the experimental design an independent measures design ?

**Fleur** yes

**System** Okay.  
You should use pearson chi square or likelihood ratio .

**Fleur** i dont understand the second test what is it

**System** Okay.  
I don't know which statistical technique you should use .

*Fleur doesn't know what likelihood ratio is, and wants to know more about it. She does so by asking what "the second test" is. The system doesn't get that "the second test" refers to likelihood ratio. Because the word "test" is used, the system matches this and again assumes the user wants to find out which statistical technique to use. The system starts executing the same plan again that it did before, but because all questions have been answered before and its answers are already in BEL the system doesn't ask these questions again. It reaches the same conclusion, but because the answer is already in COM the system doesn't share this same answer again. Instead another update rule fires that tells the user it doesn't know. This can be considered a bug in the dialogue manager because that particular update rule shouldn't have fired. What should've happened instead is that the system would have recognized the user's question and give information about likelihood ratio.*

**Fleur**      What is your favorite colour?  
**System**      Okay.  
                  I don't know what my favorite colour is .  
**Fleur**      What is your astrological sign?  
**System**      Okay.  
                  I don't know what my astrological sign is .

*The user asks two personal questions to which the system has no direct answer or plan to resolve them. Interestingly Fleur mixes American English (favorite) and American English (colour) into the same sentence. The system doesn't care and uses the same words as the user did.*

### 6.3.2 System additions

#### Addition #1

To make the system understand that *caterogrial* is a misspelling of *categorical* the next interpret unit is added:

```
interpret ([
    pattern ([ star (A) , caterogrial , star (B) ]) ,
    srai ([A, categorical ,B])
]) .
```

There are many ways in which to misspell a word. A more robust way of handling erroneous input would be to use a Levenshtein automaton to parse the user input. A Levenshtein automaton is a finite state automaton that recognizes the set of all strings with a certain maximum Levenshtein distance. The Levenshtein distance is the minimum number of single-character edits (insertions, deletions or substitutions) needed to change one word into another.

#### Addition #2

The phrase *"i dont understand the second test what is it"* lacks punctuation, capitalization and contains a spelling error. To make the system recognize this input the following units will be added.

```
interpret ([
    pattern ([ star (A) , dont , star (B) ]) ,
    srai ([A, do , not ,B])
]) .

interpret ([
    pattern ([ i , do , not , understand , the , star (A) , what ,
              is , it , star (R) ]) ,
```

```

    srai ([ what , is , the , A , ? , R ] )
  ]) .

interpret ([
  pattern ([ what , is , the , first , star ( - ) , ? , star ( R ) ])
  ,
  that ([ star ([ - , - , - ] ) , that , star ([ A , B , C ] ) , star (
    First ) , or , star ([ A , B , C ] ) , star ( - Second ) ]) ,
  meaning ([ you , ask , me , what , First , is ] ) ,
  srai ( R )
]) .

interpret ([
  pattern ([ what , is , the , second , star ( - ) , ? , star ( R )
  ]) ,
  that ([ star ([ - , - , - ] ) , that , star ([ A , B , C ] ) , star (
    - First ) , or , star ([ A , B , C ] ) , star ( Second ) ]) ,
  meaning ([ you , ask , me , what , Second , is ] ) ,
  srai ( R )
]) .

```

The first unit ensures that when a user enters a phrase that contains *dont* this part is translated to *do not*. Thanks to this first unit the phrase "*i dont understand the second test what is it*" is updated to "*i do not understand the second test what is it*". Because punctuation is missing it might not be clear that these are actually two sentences, namely "*I don't understand the second test.*" and "*What is it?*".

The second unit above matches this new phrase and rewrites it to "*what is the second test?*".

The third and fourth units match user input like "*What is the first one?*", and "*What is the second test?*". While matching the user input, the interpret pattern also looks at the contents of the *THAT* stack. If the system has just produced "*You should use pearson chi square or likelihood ratio*" the *THAT* stack will contain [i, tell, you, that, you, should, use, pearson, chi, square, or, you, should, use, likelihood, ratio]. By combining the user input and the *THAT* content the third and fourth interpret unit will yield the meaning [you, ask, me, what, pearson, chi, square, is] or [you, ask, me, what, likelihood, ratio, is]. The system can answer these questions because the *BEL* field contains facts such as [likelihood, ratio, is, a, statistical, technique].

### Addition #3

The third addition done to the system after this interaction consists of the following two facts.

```
bel([my, favorite , colour , is , infrared]).  
bel([my, astrological , sign , is , libra]).
```

After adding these two facts the system is able to answer the questions "What is your favorite colour?" and "What is your astrological sign?".

## 6.4 Demo interaction

The following is a demo interaction that shows grounding, task switching, and different forms of question accommodation.

**System** Hello.  
How can I help you ?

**User** Which statistical test should I use?

*This interaction starts the same way as the ones above. This time however we'll give some different answers to the questions the system asks.*

**System** Let's see.  
Okay.  
How many dependent variables are there ?

**User** What is a dependent variable?

*A clarification question is asked. This causes the system to push the issue [what,a,dependent,variable,is] onto the QUD stack causing the system to try to resolve this question before continuing with the current plan.*

**System** Okay.  
a dependent variable is the outcome or effect variable .  
How many dependent variables are there ?

*Because the BEL field contains a fact that resolves the top issue, this fact is output. After this the system continues with the plan by reasking how many dependent variables there are.*

**User** two

**System** Okay.  
are the dependent variables continuous ?

**User** ehm, i mean, there is one dependent variable.

*The user actually meant to say there is only one dependent variable. It is able to revise what was previously answered. The system accepts this new information and retracts the old fact from the BEL field because it is conflicting. The plan for finding out which statistical technique is reloaded which causes the system to now ask another question.*

**System** Okay.  
Is the dependent variable continuous or categorical ?



*The next question in the plan is whether the dependent variable is continuous or categorical.*

**User**        it's continuous

**System**     Okay.

How many independent variables are there ?

*The next two questions on the plan are how many independent variables there are, and if there is only one, whether this is a continuous variable. The system asks questions one by one, so first it outputs: "How many independent variables are there?"*

**User**        There is one continuous independent variable.

*One form of question accommodation allows the user to provide answers to questions that haven't been raised yet. In this case the user tells the system that there is one continuous independent variable. This answers the current question but also the next unasked one. The system is able to resolve this question without first asking it.*

**System**     Okay.

Does the data meet assumptions for parametric tests ?

*The system has one last question to resolve before it can tell the user which statistical technique to use. Assumptions for parametric tests are that the underlying source populations come from a normal distribution and that one's measures derive from an equal-interval scale. It makes sense for the user to ask the following related follow-up question.*

**User**        How can I check for normality?

*This question causes the system to load another plan and temporarily abandon the current one.*

**System**     Let's see.

Okay.

Do you want to check for normality visually or do you want to test for normality ?

*The system outputs two grounding utterances "Let's see." and "Okay." to indicate it has understood the question and is loading a new plan. Then it asks the first question on the plan it has just loaded.*

**User**        I want to test for normality.

*One can check for normality by inspecting a graph such as a histogram, P-P plot or Q-Q plot. One can also test for normality by executing a Kolmogorov-Smirnov test or a Shapiro Wilk test. The user want to use a test.*

**System**     Okay.

do you have more than 50 cases ?

**User**       ehm yes, more than 50.

*The user answers the question but uses some filler words. The system recognizes the word "yes" but only after ignoring the other input. This means that the system assumes it has understood the user but isn't entirely sure.*

**System**    Okay.  
              you have more than 50 cases .  
              you can test for normality by using the kolmogorov smirnov test.  
              Let's find out which statistical technique you should use .  
              Let's see.  
              Does the data meet assumptions for parametric tests ?

*The system uses implicit feedback ("you have more than 50 cases") to let the user know what it has understood. Now it has gathered enough information to conclude and inform the user that it can check for normality using the Kolmogorov-Smirnov test. Then it returns to the issue of finding out which statistical technique is most appropriate for the research question at hand. It reloads the previous plan which causes the system to output "Let's see". The next unresolved question on the plan deals with finding out if the data meets assumptions for parametric tests.*

**User**       yes

**System**    Okay.  
              You should use pearson correlation or regression .

*The system now informs the user of the appropriate technique to use. This removes the issue from the QUD stack and moves this fact to the COM field to indicate this is shared knowledge now.*

**User**       lhdsdskjs sdoij

**System**    I don't understand.

*The user says something of which the system can't make heads nor tails and informs the user of this.*

**User**       What time is it?

**System**    Let's see.  
              Okay.  
              the time is 24 minutes to 1 in the afternoon .

*The user asks what time it is. A lot takes place in the system before it answers this question. First, the issue [what,time,it,is] is made topmost on the QUD stack. Because no resolving fact is available in the BEL field, but there is a plan that can resolve this issue, this plan is loaded. The plan is very short but consists of an action that executes some prolog code that finds out the current time by looking at the system clock, and asserting this fact as a natural language phrase to the BEL field. Now the system is able to resolve this question and outputs the time.*

**User** eh, and tell me, if you can, what's the time in london?

**System** Do you want to know what the time in london is ?

*The system has to skip quite some input before it encounters something that matches with a known interpret pattern. The recognition rate is therefore quite low which causes the system to explicitly ask what the user wants to know.*

**User** yes

**System** Let's see.  
Okay.

the time in london is 24 minutes to 12 in the morning .

*The system can answer questions about different times in the world by making use of a REST webservice called worldclockdb. A similar thing as above happens, only now the plan doesn't contain an action to query the system clock, but it makes a call to this webservice. This results in the fact [the,time,in,london,is,24,minutes,to,12,in,the,morning] being added to the BEL field. Now the system can inform the user of the current time in London.*

**User** And Toronto?

**System** Let's see.  
Okay.

the time in toronto is 24 minutes to 8 in the morning .

*"And Toronto?" shows that the system can resolve elliptical questions. This works because there is an interpret unit that matches both the user input as the top element on the THAT stack which is in this case [i,tell,you,that,the,time,in,london,is,24,minutes,to,12,in,the,morning].*

**User** I have a fiat panda.

**System** Okay.

*The user informs the system of a fact. This fact is added as is to the BEL field.*

**User** What kind of car do i have?

**System** Okay.  
I don't know what kind of car you have .

*Although the system knows the user has a fiat panda, it doesn't know that a fiat panda is a kind of car. Therefor it doesn't know how to answer this question.*

**User** a fiat panda is a kind of car.

**System** Okay.

*The system is informed of a new fact. Now it does know that a fiat panda is a kind of car.*

**User**        What kind of car do i have?

**System**     Okay.  
                 you have a fiat panda .

*These two facts combined give the system the ability to derive new facts and answer this question.*

**User**        Am I happy?

**System**     Okay.  
                 I don't know if you are happy .

*The system doesn't know whether the user is happy or not.*

**User**        If I have a fiat panda then I am happy.

**System**     Okay.

*The user cannot only tell facts, it can also tell rules which the system happily adds to the BEL field.*

**User**        Am I happy?

**System**     Okay.  
                 you are happy .

*The system is now able to derive new facts from rules and facts it already knows.*

## Chapter 7

# Discussion and conclusion

The objective of this research has been to show by implementation how a chatbot can be equipped with a dialogue manager. This how question is answered in chapter 5. A hybrid system is described that uses a dialogue manager that works directly with natural language instead of some knowledge representation. This relieves the system of the responsibility of converting between natural language and a knowledge representation.

Chapter 4 enumerated the features and properties that such a hybrid system would yield:

1. Suitable for trivial question answering.
2. Systems can be built quickly.
3. Systems can be built by non-linguists.
4. Systems can be extended and maintained easily.
5. Can engage in small-talk.
6. Can respond to questions outside of its knowledge domain.
7. Can engage in mixed-initiative dialogues, for non-trivial question answering.
8. Uses plans to help users achieve their information goals.
9. Can deal with belief revision.
10. Outputs grounding dialogue moves.
11. Implements question accommodation.

This chapter will review these points one by one leading towards an answer to the following questions:

- How will such a hybrid system perform?
- What are its limitations?

## 7.1 Discussion

### 7.1.1 Suitable for trivial question answering

A *trivial question* refers to a question for which an answer exists in the form of a fact in the system's knowledge base and no plan execution of any kind is needed.

An example would be "*What is a dependent variable?*". This question is resolved by the fact [a,dependent,variable,is,the,outcome,or,effect,variable]. Resolve units match questions to their answers. In this case the following unit applies.

```
resolve ([
    query ([ what , star (A) ] ) ,
    match ([ star (A) , star (B) ] ) ,
    fact ([ A,B ] )
]) .
```

In an ordinary chatbot like ALICE one could have added the following category:

```
<category>
  <pattern>WHAT IS A DEPENDENT VARIABLE</pattern>
  <template>A dependent variable is exactly what its
    name suggests. It is a variable that depends on
    other variables. For example, a test score could
    be a dependent variable because it may depend on
    other variables such as a variable that takes on
    the amount of hours that you studied. Usually
    when looking for a relationship between two
    variables you try to find out what and to what
    degree one (independent) variable influences
    another dependent variable.</template>
</category>
```

This category would output a whole block of text whenever the user asks what a dependent variable is. One could in theory add the fact [a,dependent,variable,is,exactly, ...,another,dependent,variable] to the *BEL*. The *BEL* is however meant to store facts, not entire blocks of text. This makes the system only suitable for answering questions for which a one sentence answer suffices.

The hybrid system currently has no easy way of outputting more elaborate answers like in this example. An extension to the system would be necessary.

### **7.1.2 Systems can be built quickly**

When implementing a new domain, one would start by defining one or several plans. Next to plans a developer would also add some knowledge that would be initially loaded into the *BEL* set. This would be enough to get a basic system running and should take anywhere between a few minutes to a few hours. The hardest part is coming up with a good sequence of plan actions. In the statistics example a decision tree was used from a statistics book. Other ways of coming up with a plan would be to observe interactions between two humans operating within the domain of interest, for example between a statistics tutor and a student, or a traveler and a ticket-seller.

### **7.1.3 Systems can be built by non-linguists**

Implementing a new domain should be reasonable easy. By inspecting the plans of the statistics domain one shouldn't have much trouble adding new plans that can answer questions in other domains. When building a system one wouldn't have to start from scratch, since most interpret, resolve and generate units can be reused. These units however don't completely cover the English language and sooner or later units would have to be added. The best approach would be to look at the existing ones and find one that can be adapted to cover the additional subset of English.

Units can define patterns that make use of prolog's DCG formalism. Defining grammars using this formalism may take some getting used to. It is however expected that the main focus on implementing new domains will be on defining plans, and not adding many complicated units.

Someone who is a linguist and has learned about transformational grammar will see that the different units actually define how various sentence transformations happen, such as wh-movement and inversion. Knowledge of this theory is however not necessary for defining new units.

### **7.1.4 Systems can be extended and maintained easily**

The previous chapter showed three user interactions and how the dialogues were used to improve the system. These improvements consisted mostly of adding interpret, resolve or generate units or adding plans. The additions are in itself not hard to understand. The hardest part is determining what to add. How should a maintainer determine whether to add some unit, a plan or perhaps something else. This is easy if you understand how the system works, but may be harder otherwise.

### 7.1.5 Can engage in small talk

The GoDiS dialogue manager which served as the main inspiration for the hybrid system focusses on something called issue-based dialogue. This means that it views dialogue as the raising and resolving of questions. Some smalltalk can be viewed in this way as well.

**User** How are you?

**System** I am fine.

**User** What do you think of the weather?

**System** I think that the weather is very nice.

By adding the facts [i,am,fine] and [i,think,that,the,weather,is,very,nice] to the *BEL* set on startup, the system would be able to hold the dialogue above.

Small talk is not really about inquiry and there is more happening than a simple exchange of information. It can be viewed as a way of establishing some common ground and create a mini-bond with new or existing contacts.

On a surface level small talk can be simulated by adding some facts to the *BEL* set or even add a plan that raises some questions about the weather, sports or other small talk related subjects.

### 7.1.6 Can respond to questions outside of its knowledge domain

Although the example domain implements a system that can answer statistics related questions, it also responds to questions (and assertions) outside of this domain. When no input is recognized the system simply outputs "*I don't understand*". When there is a pattern that matches the input as a valid question and there is no fact in the *BEL* set or plan to resolve it, the system simply replies "*I don't know*". In contrast, a dialogue system will always respond saying it didn't understand when the user utters something that's foreign to the domain the system implements.

It is possible to make assertions which the system will add to the *BEL* set. The system can be queried for these assertions. Something like this happened in one of the user interactions.

**Mathilde** You have a girlfriend. Her name is Venna.

**System** Okay.

**Mathilde** Do you have a girlfriend?

**System** Okay.  
i have a girlfriend .

**Mathilde** What is her name?



**System**    Okay.  
             her name is venna .

This shows that the system can learn new facts, and then understand questions that could be resolved by these new facts.

### **7.1.7    Can engage in mixed-initiative dialogues, for non-trivial question answering**

*Non-trivial* questions refer to questions that can't be solved by matching with a fact in the *BEL* set. When the user asks a question and there is a plan for answering this question, this plan is loaded and executed. Although the execution of a plan causes the system to ask a series of question in a fixed order, the user may at any time divert from this. The user may provide questions to answers that haven't been raised yet, or ask a question that causes the system to load another plan and start executing that. In no way the user is forced to follow a certain sequence or to answer a question before it can ask a new question him or herself.

### **7.1.8    Uses plans to help users achieve their information goals**

Just as in GoDiS, the system developer can define plans that the system can use to resolve questions for which no direct answer exists. Several plans were added. There is one *top* plan that is used to find out what the user wants to know. There are two statistics related plans. One for finding out which statistical test is most appropriate and one for finding out how to check for normality. All these plans cause the system to ask further questions to resolve an open issue. Finally there are two plans that are used to find out what time it is either locally or somewhere in the world. These plans don't engage in some dialogue with the user but call a local or web api to find out the requested time.

### **7.1.9    Can deal with belief revision**

The following fragment, as taken from the demo interaction, shows how the user is able to revise some of the things the system believes.

**System**    How many dependent variables are there ?  
**User**        two  
**System**    Okay.  
             are the dependent variables continuous ?  
**User**        ehm, i mean, there is one dependent variable.  
**System**    Okay.  
             Is the dependent variable continuous or categorical ?

First the system is told that there are two dependent variables. But this "fact" is later removed from the *BEL* set when the user informs the system that there is only one dependent variable.

#### 7.1.10 Outputs grounding dialogue moves

We can observe from the user interactions and demo dialogue that the system produces several grounding moves.

Sequencing and grounding moves that the system outputs are:

- "*Let's see.*" - When the system loads a plan.
- "*Let's find out...*" - When the system reloads an already started plan.
- "*Okay.*" - When an assertion or question is understood.
- "*I don't understand.*" - When no input is recognized.

Furthermore, as can be seen in the demo dialogue, the system produces implicit feedback "*you have more than 50 cases .*" and explicit feedback "*Do you want to know what the time in london is ?*", depending on how much user input was matched.

#### 7.1.11 Implements question accommodation

The GoDiS system, that was used as a reference on how to implement a dialogue manager, was able to handle three main types of question accommodation. The hybrid system implements two of them.

The first kind is called **local accommodation**. Local accommodation allows the user to give elliptical answers.

Both the user interactions as the demo dialogue show several examples of this. For example the following exchange

**System** Is the dependent variable continuous or categorical ?

**Mathilde** Categorical

and

**System** How many independent variables are there ?

**Mathilde** 1

The hybrid system handles elliptical answers differently than GoDiS. In GoDiS elliptical answers are resolved by seeing if the answer resolves one of the issues on the *QUD* stack. In the hybrid system elliptical answers are resolved using interpret units that provide patterns for both the user input as the *THAT* stack. So in the hybrid system elliptical answers are resolved

by looking at what was asked previously. This is similar to how the ALICE chatbot resolves elliptical answers.

The second type of question accommodation is **global accommodation**.

Global accommodation refers to the ability to answer questions before they are asked. For example:

**System**    How many independent variables are there ?

**User**        There is one continuous independent variable.

Here the user input not only resolves the question "*How many independent variables are there?*", but also the question that was supposed to come next "*Is the independent variable continuous or categorical?*".

The third kind of question accommodation is called **dependent accommodation**. Dependent accommodation refers to the ability of the system to find out what the user wants to know based on an assertion. So when a user would say "*There is one dependent variable*" the system would be able to guess that the user wants to know which statistical technique to use and load the appropriate plan. The hybrid system doesn't currently implement dependent accommodation although adding this feature should be reasonably straightforward.

## 7.2 Conclusion

The purpose of this project was to investigate how to equip a chatbot with a dialogue manager. A prototype system was built that implements this idea. After evaluating what goes into developing and maintaining a statistics tutor, we can conclude that this chatbot/dialogue system hybrid enables us to quickly build systems with similar ease as building a regular chatbot. Furthermore, this hybrid approach yields a system that besides trivial question answering and small talk can engage the user in more complex conversations.

The dialogue manager, as was modeled after the GoDiS dialogue manager, includes the ability to deal with grounding and belief revision, something that is not found in current chatbot systems.

By sidestepping the whole problem of parsing natural language into a semantic representation, developing and extending systems has become much easier for non-linguists. In addition, the use of general patterns to match user input enables the system to converse about, or at the least properly respond to, topics that extend the domain the system was built for.

## 7.3 Further work

This section explains some of the ideas that would counter for some of the shortcomings described above.

### 7.3.1 Make domains easier to extend and maintain

For this system to be effectively applied to other domains, developers and maintainers would benefit from tutorials that show all steps that go into maintaining and extending the system. These tutorials would describe common addition that one would make such as adding plans, interpret units or facts to the *BEL* set.

The system keeps a history of all updates to the information state which makes it possible to find out which update rules were triggered and which interpret, resolve and generate units were used in a dialogue. This history can be inspected from the prolog interpreter. To ease inspection, developers and maintainers could benefit from some graphical representation of a trace that shows all user input and system output together with the update rules that were triggered, relevant units and an overview of how the information state changes during the interaction.

Furthermore, the system would be much more user friendly if developers wouldn't have to directly edit prolog code when adding new units, facts or plans to the system but could use some kind of graphical user interface be it html or otherwise.

### 7.3.2 Add some plans for engaging in small talk

On a surface level, small talk involves the exchange of trivial information. People often engage in some kind of small talk before getting down to business. An option would be to add an raise ([how,you,are]) to the top plan. This would cause the system to ask how the user is when the interaction starts. In this study no research was done on how a model for small talk could be implemented, but it would be interesting to do so and incorporate this into a future version of the system.

### 7.3.3 Improve the suitable for trivial question answering

One of the mentioned shortcomings of the system is that it can't give more elaborate answers very well. It can only output simple definitions such as in the "What is a dependent variable?" example. Instead of having the system output "*A dependent variable is the outcome or effect variable.*", we would want the system to produce the following:

A dependent variable is exactly what its name suggests. It is a variable that depends on other variables. For example, a test score could be a dependent variable because it may depend on other variables such as a variable that takes on the amount of hours that you studied. Usually when looking for a relationship between two variables you try to find out what and to what

degree one (independent) variable influences another dependent variable.

One approach would be to have plans for explaining concepts such as this one. A new plan action (in addition to plan actions like *findout*, *raise*, *ifthen*, etc.) could be added that does nothing more than outputting blobs of fixed text.

A perhaps better way would be to have a plan push new items onto the *QUD* that the system will try to answer using facts from the *BEL* set. If we would add the issues [if,a,dependent,variable,is,exactly,what,its,name,suggests] and [why,is,a,dependent,variable,exactly,what,its,name,suggests] to the *QUD* and make sure that the *BEL* set contains [a,dependent,variable,is,exactly,what,its,name,suggests,because,it,is,a,variable,that,depends,on,other,variables] it should output something similar to the first two sentences of the explanation. Either way, a new plan action would need to be added to the system to produce output like this.

#### 7.3.4 Rewrite parts of the system

The time it takes the system to respond to user input is still reasonable given that it runs on somewhat modern and fast hardware. Adding many facts to the *BEL* field slows the system down quite a bit. The system would benefit from a rewrite of some of its parts. Especially the matching algorithm could use improvement since this is where the system spends the majority of execution time.

#### 7.3.5 Allow for action-oriented dialogue

The system only allows for issue-oriented dialogue. This means that its purpose is to answer questions. It won't respond to commands or requests unless this request entails answering a question. To allow for action-oriented dialogue as well, the information state would have to include a queue especially for actions. GoDiS already contains a queue like this called *ACTIONS* in its information state. Some new update rules would have to be added as well, but again GoDiS can serve as an example on what these could look like.

#### 7.3.6 Further ideas

So far only a very basic statistics tutor has been implemented and some plans have been added to get the time both locally as around the world. One can think of many interesting domains to add to the system. It has been shown that the system can call web services which brings to mind many interesting applications such as the ability to answer questions about the weather or train schedules.

A speech recognition and speech synthesis component could be added to the system to make it speak and respond to spoken input. Depending on the domain the system could be embedded into cars, robots or perhaps toys.

The demo dialogue shows how the user can enter if/then rules and facts. It is shown that the system has limited reasoning abilities. This could be turned into a system that accepts business rules and can also execute them. The system could be extended to produce grounding feedback when conflicting or unclear rules and facts are entered.

If the system is embedded into, for instance, a robot, the robot might be programmed using natural language. One would be able to tell the robot things such as: "If sensor 1 is activated then start rotating motor 4 in the opposite direction" or "If sensor 1 is activated then you've hit a wall".

The dialogue manager itself could be extended to be able to deal with multi-modal input and output. Other improvements would include the ability to have a conversation with more than one person and produce instructional dialogue which would be very useful for applications in education.

# Appendix A

## statistics.pl

```
% statistics.pl
% Aswin van Woudenberg

% how to check for NORMALITY
interpret([
    pattern([star(_),how,star(_),normality,star(R)
    ]),
    meaning([you,ask,me,how,you,can,check,for,
    normality]),
    srai(R)
]).

% how to see if data is NORMALLY distributed
interpret([
    pattern([star(_),how,star(_),normally,star(R)
    ]),
    meaning([you,ask,me,how,you,can,check,for,
    normality]),
    srai(R)
]).

% which TEST should i use? / HOW I CAN HELP YOU
interpret([
    pattern([star(_),test,star(_),star(R)]),
    that([i,ask,you,how,i,can,help,you]),
    srai([which,statistical,technique,should,i,use
    ,?,R])
]).

% VISUALLY please / if you want to check for normality
```

```

        visually or if you want to test for normality
interpret ([
    pattern ([star (-), visually, star (R)]) ,
    that ([i, ask, you, if, you, want, to, check, for,
        normality, visually, or, if, you, want, to, test,
        for, normality]) ,
    meaning ([you, tell, me, that, you, want, to, check,
        for, normality, visually]) ,
    srai (R)
]) .

```

```

% i'd like to use a TEST / if you want to check for
% normality visually or if you want to test for
% normality
interpret ([
    pattern ([star (-), test, star (R)]) ,
    that ([i, ask, you, if, you, want, to, check, for,
        normality, visually, or, if, you, want, to, test,
        for, normality]) ,
    meaning ([you, tell, me, that, you, want, to, test, for
        , normality]) ,
    srai (R)
]) .

```

```

% WHAT is a DEPENDENT variable?
interpret ([
    pattern ([star (-), what, star (-), dependent, star (-
        )]) ,
    srai ([what, is, a, dependent, variable, ?])
]) .

```

```

% WHAT does CONTINUOUS mean?
interpret ([
    pattern ([star (-), what, star (-), continuous, star (
        -)]) ,
    srai ([what, is, a, continuous, variable, ?])
]) .

```

```

% WHAT does CATEGORICAL mean?
interpret ([
    pattern ([star (-), what, star (-), categorical, star
        (-)]) ,
    srai ([what, is, a, categorical, variable, ?])
]) .

```



```

% HELLO.
generate ([
    agenda (greet , []) ,
    template ([ 'Hello . ' ]) ,
    meaning ([ i , greet , you ])
]) .

% HOW YOU CAN CHECK FOR NORMALITY / YOU CAN TEST FOR
  NORMALITY BY USING the shapiro wilk test / YOU CAN
  TEST FOR NORMALITY BY USING the shapiro wilk test
resolve ([
    query ([how , you , can , check , for , normality ]) ,
    match ([you , can , test , for , normality , by , using ,
        star (A) ]) ,
    fact ([you , can , test , for , normality , by , using , A])
]) .

% Dialogue plans

plan (top ,
[
    raise ([how , i , can , help , you ]) ,
    findout ([ if , you , want , to , know , which , statistical ,
        technique , you , should , use ])
]) .

plan ([how , you , can , check , for , normality ] ,
[
    findout ([ if , you , want , to , check , for , normality , visually
        , or , if , you , want , to , test , for , normality ]) ,
    ifthenelse ([you , want , to , check , for , normality , visually
        ] ,
    [ add ([you , can , check , for , normality , by , inspecting , the
        , histogram , or ,
            you , can , check , for , normality , by , inspecting , the ,
            p , p , plot ]) ] ,
    [ findout ([ if , you , have , more , than , 50 , cases ]) ,
        ifthenelse ([you , have , more , than , 50 , cases ] ,
        [add ([you , can , test , for , normality , by , using , the ,
            kolmogorov , smirnov , test ]) ] ,
        [add ([you , can , test , for , normality , by , using , the ,
            shapiro , wilk , test ]) ] )
    ])
])

```

)).

```

plan ([ which , statistical , technique , you , should , use ] ,
[
  findout ([ how , many , dependent , variables , there , are ] ) ,
  ifthenelse ([ there , is , one , dependent , variable ] ,
  [
    findout ([ if , the , dependent , variable , is , continuous ,
      or , if , the , dependent , variable , is , categorical ] ) ,
    ifthenelse ([ the , dependent , variable , is , continuous ] ,
    [
      findout ([ how , many , independent , variables , there ,
        are ] ) ,
      ifthenelse ([ there , is , one , independent , variable ] ,
      [
        findout ([ if , the , independent , variable , is ,
          continuous , or , if , the , independent , variable ,
          is , categorical ] ) ,
        ifthenelse ([ the , independent , variable , is ,
          categorical ] ,
        [
          findout ([ if , the , independent , variable , is ,
            dichotomous ] ) ,
          ifthenelse ([ the , independent , variable , is ,
            dichotomous ] ,
          [
            findout ([ if , the , experimental , design , is , an ,
              independent , measures , design ,
              or , if , the , experimental , design , is ,
              a , repeated , measures , design ] ) ,
            ifthenelse ([ the , experimental , design , is , an ,
              independent , measures , design ] ,
            [
              findout ([ if , the , data , meets , assumptions ,
                for , parametric , tests ] ) ,
              ifthenelse ([ the , data , meets , assumptions ,
                for , parametric , tests ] ,
              [ add ([ you , should , use , the , independent , t ,
                test , or , you , should , use , the , point ,
                biserial , correlation ] ) ] ,
              [ add ([ you , should , use , the , mann , whitney ,
                test ] ) ] )
            ] ,
            [

```

```

        findout ([ if , the , data , meets , assumptions ,
                    for , parametric , tests ] ) ,
        ifthenelse ([ the , data , meets , assumptions ,
                    for , parametric , tests ] ,
        [ add ([ you , should , use , the , dependent , t ,
                    test ] ) ] ,
        [ add ([ you , should , use , the , wilcoxon ,
                    matched , pairs , test ] ) ] )
    ])
],
[
    findout ([ if , the , experimental , design , is , an ,
                independent , measures , design ,
                or , if , the , experimental , design , is ,
                a , repeated , measures , design ] ) ,
    ifthenelse ([ the , experimental , design , is , an ,
                independent , measures , design ] ,
    [
        findout ([ if , the , data , meets , assumptions ,
                    for , parametric , tests ] ) ,
        ifthenelse ([ the , data , meets , assumptions ,
                    for , parametric , tests ] ,
        [ add ([ you , should , use , one , way , independent
                    , anova ] ) ] ,
        [ add ([ you , should , use , the , kruskal , wallis ,
                    test ] ) ] )
    ] ,
    [
        findout ([ if , the , data , meets , assumptions ,
                    for , parametric , tests ] ) ,
        ifthenelse ([ the , data , meets , assumptions ,
                    for , parametric , tests ] ,
        [ add ([ you , should , use , one , way , repeated ,
                    measures , anova ] ) ] ,
        [ add ([ you , should , use , friedman , '\ ' , s ,
                    anova ] ) ] )
    ])
])
],
[
    findout ([ if , the , data , meets , assumptions , for ,
                parametric , tests ] ) ,
    ifthenelse ([ the , data , meets , assumptions , for ,
                parametric , tests ] ,

```

```

[add([you, should, use, pearson, correlation, or,
      you, should, use, regression])],
[add([you, should, use, spearman, correlation, or,
      you, should, use, kendall, '\', s, tau])])
])
],
[
  findout([if, the, independent, variables, are,
            continuous,
            or, if, the, independent, variables, are,
            categorical,
            or, if, there, are, continuous, and,
            categorical, independent, variables
            ]),
  ifthenelse([the, independent, variables, are,
               categorical],
  [
    findout([if, the, experimental, design, is, an,
              independent, measures, design,
              or, if, the, experimental, design, is, a,
              repeated, measures, design,
              or, if, the, experimental, design, is, a,
              mixed, model, design]),
    ifthenelse([the, experimental, design, is, an,
                 independent, measures, design],
  [
    findout([if, the, data, meets, assumptions, for,
              parametric, tests]),
    ifthen([the, data, meets, assumptions, for,
             parametric, tests],
    [add([you, should, use, independent, factorial,
           anova, or, you, should, use, multiple,
           regression])])
  ],
  [
    ifthenelse([the, experimental, design, is, a,
                 repeated, measures, design],
  [
    findout([if, the, data, meets, assumptions,
              for, parametric, tests]),
    ifthen([the, data, meets, assumptions, for,
             parametric, tests],
    [add([you, should, use, independent,
           factorial, repeated, measures, anova])])
  ]
  ]
  ]
  ]

```

```

    ],
    [
        findout ([ if , the , data , meets , assumptions ,
                    for , parametric , tests ] ) ,
        ifthen ([ the , data , meets , assumptions , for ,
                    parametric , tests ] ,
                [add ([you , should , use , factorial , mixed ,
                        anova])])
    ])
])
],
[
    findout ([ if , the , independent , variables , are ,
                continuous ] ) ,
    ifthenelse ([ the , independent , variables , are ,
                continuous ] ,
                [
                    findout ([ if , the , data , meets , assumptions , for
                                , parametric , tests ] ) ,
                    ifthen ([ the , data , meets , assumptions , for ,
                                parametric , tests ] ,
                            [add ([you , should , use , multiple , regression])
                                ])
                ]
    ],
    [
        findout ([ if , the , data , meets , assumptions , for
                    , parametric , tests ] ) ,
        ifthen ([ the , data , meets , assumptions , for ,
                    parametric , tests ] ,
                [add ([you , should , use , multiple , regression ,
                        or , you , should , use , ancova])])
    ])
])
],
[
    findout ([ how , many , independent , variables , there ,
                are ] ) ,
    ifthenelse ([ there , is , one , independent , variable ] ,
                [
                    findout ([ if , the , independent , variable , is ,
                                continuous , or , if , the , independent , variable ,
                                is , categorical ] ) ,
                    ifthenelse ([ the , independent , variable , is ,

```

```

        categorical],
    [
        findout([if, the, experimental, design, is, an,
            independent, measures, design]),
        ifthen([the, experimental, design, is, an,
            independent, measures, design],
            [add([you, should, use, pearson, chi, square, or,
                you, should, use, likelihood, ratio])])
    ],
    [add([you, should, use, logistic, regression, or,
        you, should, use, biserial, correlation, or, you,
        should, use, point, biserial, correlation])])
],
[
    findout([if, the, independent, variables, are,
        continuous,
        or, if, the, independent, variables, are,
        categorical,
        or, if, there, are, continuous, and,
        categorical, independent, variables
        ]),
    ifthenelse([the, independent, variables, are,
        categorical],
    [
        findout([if, the, experimental, design, is, an,
            independent, measures, design]),
        ifthen([the, experimental, design, is, an,
            independent, measures, design],
            [add([you, should, use, loglinear, analysis])])
    ],
    [
        ifthenelse([the, independent, variables, are,
            continuous],
            [add([you, should, use, logistic, regression])],
            [
                findout([if, the, experimental, design, is, an,
                    independent, measures, design]),
                ifthen([the, experimental, design, is, an,
                    independent, measures, design],
                    [add([you, should, use, logistic, regression])
                    ])
            ])
    ])
])
])
])

```

```

    ])
  ],
  [
    findout ([ if , the , dependent , variables , are , continuous
              ] ) ,
    ifthen ([ the , dependent , variables , are , continuous ] ,
    [
      findout ([ how , many , independent , variables , there ,
                 are ] ) ,
      ifthenelse ([ there , is , one , independent , variable ] ,
      [
        findout ([ if , the , independent , variable , is ,
                   categorical ] ) ,
        ifthen ([ the , independent , variable , is ,
                  categorical ] ,
        [
          findout ([ if , the , data , meets , assumptions , for ,
                     parametric , tests ] ) ,
          ifthen ([ the , data , meets , assumptions , for ,
                   parametric , tests ] ,
          [ add ([ you , should , use , manova ] ) ] )
        ] )
      ] )
    ] ,
    [
      findout ([ if , the , independent , variables , are ,
                 categorical ,
                 or , if , there , are , continuous , and ,
                 categorical , independent , variables
                 ] ) ,
      ifthenelse ([ the , independent , variables , are ,
                    categorical ] ,
      [
        findout ([ if , the , data , meets , assumptions , for ,
                   parametric , tests ] ) ,
        ifthen ([ the , data , meets , assumptions , for ,
                 parametric , tests ] ,
        [ add ([ you , should , use , factorial , manova ] ) ] )
      ] ,
      [
        findout ([ if , the , data , meets , assumptions , for ,
                   parametric , tests ] ) ,
        ifthen ([ the , data , meets , assumptions , for ,
                 parametric , tests ] ,
        [ add ([ you , should , use , mancova ] ) ] )
      ]
    ]
  ]

```

```

    ])
  ])
])
])
]) .

bel ([ the , independent , t , test , is , a , statistical , technique
    ]) .
bel ([ point , biserial , correlation , is , a , statistical ,
    technique ]) .
bel ([ the , mann , whitney , test , is , a , statistical , technique
    ]) .
bel ([ the , dependent , t , test , is , a , statistical , technique ])
.
bel ([ the , wilcoxon , matched , pairs , test , is , a , statistical ,
    technique ]) .
bel ([ one , way , independent , anova , is , a , statistical ,
    technique ]) .
bel ([ the , kruskal , wallis , test , is , a , statistical ,
    technique ]) .
bel ([ one , way , repeated , measures , anova , is , a , statistical ,
    technique ]) .
bel ([ friedman , '\ ' , s , anova , is , a , statistical , technique
    ]) .
bel ([ pearson , correlation , is , a , statistical , technique ]) .
bel ([ regression , is , a , statistical , technique ]) .
bel ([ spearman , correlation , is , a , statistical , technique ])
.
bel ([ kendall , '\ ' , s , tau , is , a , statistical , technique ]) .
bel ([ independent , factorial , anova , is , a , statistical ,
    technique ]) .
bel ([ multiple , regression , is , a , statistical , technique ]) .
bel ([ independent , factorial , repeated , measures , anova , is ,
    a , statistical , technique ]) .
bel ([ factorial , mixed , anova , is , a , statistical , technique
    ]) .
bel ([ ancova , is , a , statistical , technique ]) .
bel ([ pearson , chi , square , is , a , statistical , technique ]) .
bel ([ likelihood , ratio , is , a , statistical , technique ]) .
bel ([ logistic , regression , is , a , statistical , technique ]) .
bel ([ biserial , correlation , is , a , statistical , technique ])
.
bel ([ point , biserial , correlation , is , a , statistical ,
    technique ]) .

```



```

bel([loglinear ,analysis ,is ,a ,statistical ,technique]).
bel([manova,is ,a ,statistical ,technique]).
bel([factorial ,manova,is ,a ,statistical ,technique]).
bel([mancova,is ,a ,statistical ,technique]).
bel([a,categorical ,variable ,is ,a ,variable ,that ,can ,
    take,on,a ,limited ,number,of ,values]).
bel([a,continuous ,variable ,is ,a ,variable ,that ,can ,have
    ,an,infinite ,number,of ,values ,between,two ,points]).
bel([a,dichotomous ,variable ,is ,a ,variable ,which ,has ,
    only,two ,categories]).
bel([a,dependent ,variable ,is ,the ,outcome ,or ,effect ,
    variable]).
bel([an,independent ,variable ,is ,the ,variable ,that ,is ,
    changed,to ,test ,the ,effect ,on,a ,dependent ,variable
    ]).

```

## Appendix B

### datetime.pl

```
% datetime.pl
% Aswin van Woudenberg

time(H,M) —> minutes(M) , hours(H,M) , timeofday(H,M) .

timeofday(0,0) —> [midnight] .
timeofday(H,M) —> { H < 12 }, [in , the , morning] .
timeofday(H,M) —> { H >= 12 , H < 17 }, [in , the ,
    afternoon] .
timeofday(H,M) —> { H >= 17 , H < 20 }, [in , the ,
    evening] .
timeofday(H,M) —> { H >= 20 }, [at , night] .

hours(H,0) —> { H <= 12 }, [H,o,'\'',clock] .
hours(H,0) —> { H > 12 , N is H - 12 }, [N,o,'\'',
    clock] .
hours(H,M) —> { M <= 30 , M > 0 , H <= 12 }, [H] .
hours(H,M) —> { M <= 30 , M > 0 , H > 12 , N is H - 12
    }, [N] .
hours(H,M) —> { M > 30 , H < 12 , N is H + 1 }, [N] .
hours(12,M) —> { M > 30 }, [1] .
hours(H,M) —> { M > 30 , H > 12 , N is H - 11 }, [N] .

minutes(0) —> [] .
minutes(15) —> [a , quarter , past] .
minutes(30) —> [half , past] .
minutes(45) —> [a , quarter , to] .
minutes(Min) —> { member(Min,[5 , 10 , 20 , 25]) }, [Min ,
    past] .
minutes(Min) —> { member(Min,[35 , 40 , 50 , 55]) , M is 60
```

```

    - Min }, [M,to].
minutes(Min) —> { Min < 30, not(member(Min
    ,[0,5,10,20,25])) }, [Min,minutes,past].
minutes(Min) —> { Min > 30, not(member(Min
    ,[35,40,50,55])), M is 60 - Min }, [M,minutes,to].

city —> { place_zone(C,-) }, C.

place_zone([london], 'Europe/London').
place_zone([toronto], 'America/Toronto').
place_zone([austria], 'Europe/Vienna').

% AND LONDON? / THE TIME IN toronto IS half past seven
interpret([
    pattern([and,syntax(city,C,[]),?,star(R)]),
    that([i,tell,you,that,the,time,in,star(-),is,
        star(-)]),
    meaning([you,ask,me,what,the,time,in,C,is]),
    srai(R)
]).

plan([what,the,time,is],
[
    prolog((get_time(T), stamp_date_time(T,date(-,-,-,H,
        M,-,-,-,-),'local'), time(H,M,P,[]))),
    add([the,time,is,P])
]).

plan([what,the,time,in,star(Place),is],
[
    prolog((place_zone(Place,Zone), timezonedb(Zone,
        date(-,-,-,H,M,-,-,-,-)), time(H,M,P,[]))),
    add([the,time,in,Place,is,P])
]).

plan([what,the,year,is],
[
    prolog((get_time(T), stamp_date_time(T,date(Y,-,-,-,
        -,-,-,-,-),'local'))),
    add([the,year,is,Y])
]).

```

## Appendix C

### timezonedb.pl

```
% timezonedb.pl
% Aswin van Woudenberg

% http://timezonedb.com/

:- use_module(library('http/http_client')).
:- use_module(library('http/json')).

timezonedb_key('SECRET_APIKEY').
timezonedb_url('http://api.timezonedb.com').

timezonedb(Zone, Date) :-
    format(atom(Params), 'zone=~s', [Zone]),
    timezonedb_get(Params, Date).

timezonedb(Lat, Lng, Date) :-
    format(atom(Params), 'lat=~w&lng=~w', [Lat, Lng]),
    timezonedb_get(Params, Date).

timezonedb_get(Params, date(Y,M,D,H,Mn,S, GmtOffset, Abbr,
    ,DST)) :-
    timezonedb_key(Key),
    timezonedb_url(URL),
    format(atom(HREF), '~s/?~s&format=json&key=~s',
        [URL, Params, Key]),
    http_get(HREF, Json, []),
    atom_json_term(Json, json(Set), []),
    subset([status='OK', abbreviation=Abbr,
        gmtOffset=GMTOffset, dst=Dst, timestamp=
```

```

Timestamp], Set),
(Dst = '1' -> DST = true; DST = -),
term_to_atom(GmtOffset, GMTOffset),
stamp_date_time(Timestamp, date(Y,M,D,H,Mn,S,-,
-, -), 0).

```

# Bibliography

- [1] M. Mauldin, “Chatterbots, tinymuds, and the turing test: Entering the loebner prize competition,” in *Proceedings of the National Conference on Artificial Intelligence*, pp. 16–16, JOHN WILEY & SONS LTD, 1994.
- [2] B. A. Shawar and E. Atwell, “Using dialogue corpora to train a chatbot,” in *Proceedings of the Corpus Linguistics 2003 conference*, pp. 681–690, 2003.
- [3] J. Huang, M. Zhou, and D. Yang, “Extracting chatbot knowledge from online discussion forums,” in *Proceedings of the 20th international joint conference on Artificial intelligence*, pp. 423–428, Morgan Kaufmann Publishers Inc., 2007.
- [4] R. Pirrone, G. Russo, V. Cannella, and D. Peri, “Gaiml: A new language for verbal and graphical interaction in chatbots,” *Mobile Information Systems*, vol. 4, no. 3, pp. 195–209, 2008.
- [5] “Chatbots.org website,” Nov. 2012. <http://www.chatbots.org>.
- [6] “Nederlandse Energie Maatschappij website,” Nov. 2012. <http://www.nle.nl>.
- [7] “IKEA website,” Nov. 2012. <http://www.ikea.com/nl/nl/>.
- [8] “Werken bij Action website,” Nov. 2012. <http://www.werkenbijaction.nl>.
- [9] T. Bickmore, “A computational model of small talk,” *Unpublished final project in Discourse and Dialogue for Interactive Systems(MAS 962)*, Massachusetts Institute of Technology, Fall, 1999.
- [10] J. Weizenbaum, “ELIZA—a computer program for the study of natural language communication between man and machine,” *Communications of the ACM*, vol. 9, no. 1, pp. 36–45, 1966.

- [11] D. Jurafsky, J. H. Martin, and A. Kehler, *Speech and language processing: an introduction to natural language processing, computational linguistics and speech recognition*, vol. 2. MIT Press, 1999.
- [12] Xu, Wei and Rudnicky, Alexander I., “Task-based dialog management using an agenda,” in *Proceedings of the 2000 ANLP/NAACL Workshop on Conversational systems - Volume 3*, ANLP/NAACL-ConvSyst ’00, (Stroudsburg, PA, USA), pp. 42–47, Association for Computational Linguistics, 2000.
- [13] A. Gorin, G. Riccardi, and J. Wright, “How may I help you?,” *Speech Communication*, vol. 23, no. 12, pp. 113 – 127, 1997.
- [14] S. Larsson and D. Traum, “Information state and dialogue management in the TRINDI dialogue move engine toolkit,” *Natural language engineering*, vol. 6, no. 3 & 4, pp. 323–340, 2000.
- [15] D. Traum and S. Larsson, “The information state approach to dialogue management,” *Current and new directions in discourse and dialogue*, pp. 325–353, 2003.
- [16] R. Wallace, “The anatomy of ALICE,” *Parsing the Turing Test*, pp. 181–210, 2009.
- [17] A. Turing, “Computing machinery and intelligence,” *Mind*, vol. 59, no. 236, pp. 433–460, 1950.
- [18] J. Searle *et al.*, “Minds, brains, and programs,” *Behavioral and brain sciences*, vol. 3, no. 3, pp. 417–457, 1980.
- [19] J. Weizenbaum, “Computer power and human reason: From judgment to calculation.,” 1976.
- [20] D. Bobrow, “A Turing Test Passed,” *ACM SIGART Newsletter*, pp. 14–15, 1968.
- [21] K. Colby, S. Weber, and F. Hilf, “Artificial paranoia,” *Artificial Intelligence*, vol. 2, no. 1, pp. 1–25, 1971.
- [22] K. Colby, “Modeling a paranoid mind,” *Behavioral and Brain Sciences*, vol. 4, no. 4, pp. 515–560, 1981.
- [23] J. Heiser, K. Colby, W. Faught, and R. Parkison, “Can psychiatrists distinguish a computer simulation of paranoia from the real thing?: The limitations of turing-like tests as measures of the adequacy of simulations,” *Journal of Psychiatric Research*, vol. 15, no. 3, pp. 149–162, 1979.
- [24] B. J. Copeland, *Artificial intelligence*. Blackwell Oxford, 1993.

- [25] R. Epstein, “The quest for the thinking computer,” *AI magazine*, vol. 13, no. 2, p. 81, 1992.
- [26] S. Shieber, “Lessons from a restricted Turing test,” *arXiv preprint cmp-lg/9404002*, 1994.
- [27] J. Hutchens, “How to pass the Turing test by cheating,” *School of Electrical, Electronic and Computer Engineering research report TR97-05*. Perth: University of Western Australia, 1996.
- [28] D. Hofstadter, *Fluid concepts and creative analogies: Computer models of the fundamental mechanisms of thought*. Basic Books, 1996.
- [29] “C/C++ implementation of ALICE,” Jan. 2013. <http://igad.nhtv.nl/~bikker/>.
- [30] “PHP implementation of ALICE,” Jan. 2013. <http://blog.program-o.com/>.
- [31] R. Wallace, H. Tomabechi, and D. Aimless, “Chatterbots Go Native: Considerations for an eco-system fostering the development of artificial life forms in a human world,” *Joho Shori Gakkai Zenkoku Taikai Koen Ronbunshu*, vol. 65, no. 5, pp. 5–451, 2003.
- [32] S. Larsson, P. Ljunglöf, R. Cooper, E. Engdahl, and S. Ericsson, “GoDiS: an accommodating dialogue system,” in *Proceedings of the 2000 ANLP/NAACL Workshop on Conversational systems-Volume 3*, pp. 7–10, Association for Computational Linguistics, 2000.
- [33] J. Austin, *How to do things with words*, vol. 88. Harvard University Press, 1975.
- [34] J. Searle, *A taxonomy of illocutionary acts*. Linguistic Agency University of Trier, 1976.
- [35] R. Stalnaker, “Assertion. In (P. Cole, ed.),” *Syntax and Semantics: Pragmatics*, vol. 9, 1978.
- [36] H. Clark and E. Schaefer, “Contributing to discourse,” *Cognitive science*, vol. 13, no. 2, pp. 259–294, 1989.
- [37] H. Grice, “Logic and conversation,” 1975, pp. 41–58, 1975.
- [38] A. Gimson and A. Cruttenden, “Pronunciation of English,” *London: Arnold (5 th edn, revised by A. Cruttenden)*, 1994.
- [39] R. Donovan, “Trainable speech synthesis,” *Univ. Eng. Dept*, p. 164, 1996.



- [40] S. Miller, R. Bobrow, R. Ingria, and R. Schwartz, “Hidden understanding models of natural language,” in *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*, pp. 25–32, Association for Computational Linguistics, 1994.
- [41] S. Seneff, “TINA: A natural language system for spoken language applications,” *Computational linguistics*, vol. 18, no. 1, pp. 61–86, 1992.
- [42] R. Pieraccini, E. Levin, and C. Lee, “Stochastic representation of conceptual structure in the ATIS task,” in *Proceedings of the, 1991 Speech and Natural Language Workshop*, pp. 121–124, 1991.
- [43] S. Miller, D. Stallard, R. Bobrow, and R. Schwartz, “A fully statistical approach to natural language interfaces,” in *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pp. 55–61, Association for Computational Linguistics, 1996.
- [44] S. Miller, H. Fox, L. Ramshaw, and R. Weischedel, “A novel use of statistical parsing to extract information from text,” in *ACM International Conference Proceeding Series*, vol. 4, pp. 226–233, 2000.
- [45] S. Channarukul, “Yag: a template-based natural language generator for real-time systems,” 1999.
- [46] M. Walker and O. Rambow, “Spoken language generation,” 2002.
- [47] M. McTear, “Spoken dialogue technology: enabling the conversational user interface,” *ACM Computing Surveys (CSUR)*, vol. 34, no. 1, pp. 90–169, 2002.
- [48] J. Allen, D. Byron, M. Dzikovska, G. Ferguson, L. Galescu, and A. Stent, “Toward conversational human-computer interaction,” *AI magazine*, vol. 22, no. 4, p. 27, 2001.
- [49] J. Chu-Carroll, “Form-based reasoning for mixed-initiative dialogue management in information-query systems,”
- [50] J. Sturm, E. den Os, and L. Boves, “Dialogue management in the Dutch ARISE train timetable information system,” in *Proc. Eurospeech*, pp. 1419–1422, 1999.
- [51] V. Zue, S. Seneff, J. Glass, J. Polifroni, C. Pao, T. Hazen, and L. Hetherington, “JUPITER: a telephone-based conversational interface for weather information,” *Speech and Audio Processing, IEEE Transactions on*, vol. 8, no. 1, pp. 85–96, 2000.
- [52] J. Allen and C. Perrault, “Analyzing intention in utterances,” *Artificial intelligence*, vol. 15, no. 3, pp. 143–178, 1980.

- [53] D. Appelt, *Planning english sentences*. Cambridge University Press, 1992.
- [54] P. Cohen and C. Perrault, “Elements of a plan-based theory of speech acts,” *Cognitive science*, vol. 3, no. 3, pp. 177–212, 1979.
- [55] P. Cohen, “Dialogue modeling,” in *Survey of the state of the art in Human Language Technology*, pp. 204–210, Cambridge University Press, 1997.
- [56] G. Ferguson, J. Allen, *et al.*, “TRIPS: An integrated intelligent problem-solving assistant,” in *Proceedings of the National Conference on Artificial Intelligence*, pp. 567–573, JOHN WILEY & SONS LTD, 1998.
- [57] G. Ferguson, J. Allen, B. Miller, *et al.*, “TRAINS-95: Towards a mixed-initiative planning assistant,” in *Proceedings of the Third Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pp. 70–77, Edinburgh, Scotland, 1996.
- [58] M. Bratman, D. Israel, and M. Pollack, “Plans and resource-bounded practical reasoning,” *Computational intelligence*, vol. 4, no. 3, pp. 349–355, 2007.
- [59] D. Traum *et al.*, “Conversational Agency: The Trains-93 Dialogue Manager,” in *In Susann LuperFoy, Anton Nijholt, and Gert Veldhuijzen van Zanten, editors, Proceedings of Twente Workshop on Language Technology, TWLT-II*, Citeseer, 1996.
- [60] S. Larsson, A. Berman, J. Hallenborg, and D. Hjelm, “TrindiKit 3.1 Manual,” *Department of Linguistics, Göteborg University. Version June*, vol. 3, p. 2004, 2004.
- [61] J. Bos and M. Gabsdil, “First-order inference and the interpretation of questions and answers,” *Proceedings of Gotolog*, pp. 43–50, 2000.
- [62] C. Matheson, M. Poesio, and D. Traum, “Modelling grounding and discourse obligations using update rules,” in *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*, pp. 1–8, Association for Computational Linguistics, 2000.
- [63] M. Core, J. Moore, and C. Zinn, “Supporting constructive learning with a feedback planner,” in *AAAI fall symposium on building dialogue systems for tutorial applications*, 2000.
- [64] I. Lewin, “The autoroute dialogue demonstrator,” tech. rep., Citeseer, 1998.

- [65] S. Larsson and S. Ericsson, “GoDiS - issue-based dialogue management in a multi-domain, multi-language dialogue system,” in *Demonstration Abstracts, ACL-02*, 2002.
- [66] C. Zinn, J. Moore, and M. Core, “Intelligent information presentation for tutoring systems,” *Multimodal intelligent information presentation*, pp. 227–252, 2005.
- [67] J. Amores, A. Berman, J. Bos, J. Boye, R. Cooper, S. Ericsson, A. Holt, S. Larsson, D. Milward, and J. Quesada, “Knowledge and Action Management in the Home Device Environment,” 2001.
- [68] S. Larsson and J. Villing, “The dico project: A multimodal menu-based in-vehicle dialogue system,” in *Proceedings of the 7th International Workshop on Computational Semantics (IWCS-7)*, p. 4, 2007.
- [69] S. Larsson and J. Villing, “Multimodal menu-based dialogue in dico ii,” *DiaHolmia*, p. 123, 2009.
- [70] C. Loehr, “A Flexible Architecture for a Multimodal Robot Control Interface,” 2002.
- [71] J. Bos, E. Klein, O. Lemon, and T. Oka, “DIPPER: Description and formalisation of an information-state update dialogue system architecture,” in *4th SIGdial Workshop on Discourse and Dialogue*, pp. 115–124, 2003.
- [72] L. Quintal and P. Sampaio, “Prototyping Dialogues with Midiki: An Information State Update Dialogue Manager,” in *Text, Speech and Dialogue*, pp. 363–370, Springer, 2009.
- [73] S. Larsson, *Issue-based dialogue management*. Department of Linguistics, Göteborg University, 2002.
- [74] R. San-Segundo, J. M. Montero, J. Gutierrez, A. Gallardo, J. D. Romeral, and J. M. Pardo, “A telephone-based railway information system for spanish: Development of a methodology for spoken dialogue design,” in *Proceedings of the Second SIGdial Workshop on Discourse and Dialogue-Volume 16*, pp. 1–9, Association for Computational Linguistics, 2001.
- [75] J. Allwood, “An activity based approach to pragmatics,” *Abduction, belief and context in dialogue: Studies in computational pragmatics*, pp. 47–80, 2000.
- [76] A. Colmerauer and P. Roussel, “The birth of prolog,” in *History of programming languages—II*, pp. 331–367, ACM, 1996.

- [77] M. Eisenstadt and M. Brayshaw, “Build your own knowledge engineering toolkit,” tech. rep., Technical report, Human Cognition Research Laboratory, The Open University, UK, 1990.
- [78] A. Field, *Discovering Statistics Using SPSS*. SAGE Publications, 2005.