

Adventurous Linguistics

- improving adventure game dialogues by linguistic means

Master's Thesis

**Lena Carlsson
Jenny Gustafson**

Göteborg University, September 2000
Supervisor: Patricija Jaksetic, Interactive Institute

Abstract

This thesis aims to enhance today's hard-wired interactions with adventure games. Players of these games have no opportunity to influence their own contributions in the conversations with other game characters and the answers to their questions are always the same. By using linguistic knowledge, such as definition of topic, natural language generation and the semantic relationships between words, this thesis provides an alternative game interface, where the player herself gets to formulate the questions and where the answers given by the game character always differ in form. Our interface is divided in three main modules: input analysis, knowledge representation and sentence generation, all of which are implemented in the programming language Prolog. The input analysis module uses a part of speech tagger, searching for possible topics and information on what kind of question is stated. The information on the game situation is stored in, and retrieved from, a knowledge base whose size can be kept small as it has access to WordNet. The sentence generation module uses lexicon, grammar and generation rules providing every input question with an English answer originating from the content of the knowledge base. By analysing input and generating output, we contribute to a more interactive, interesting and varied game experience.

We would like to thank the following people for their help and encouragement:

Patricija Jaksetic, Interactive Institute, Göteborg University

Tobias Skog, Interactive Institute, Göteborg University

Robert Andersson, Department of Linguistics, Göteborg University

Åsa Wengelin, Department of Linguistics, Göteborg University

Per Mildner, SICS

Jesper Eskilsson, SICS

Oliver Mason, Department of English, University of Birmingham

1 Introduction

In today's adventure games typically there is one main character, whose role is taken on by the player of the game. The main character's tasks and goals can vary depending on which type of adventure game it is. Although, most games have in common that the main character has to solve a set of problems and converse with a number of other characters in order to carry out every task that leads up to the solution of the game, which not seldom means saving the world from Armageddon. In the encounter with other game characters, the player needs to interact with these individuals, to get clues about what to do next, in order to solve a particular problem. This interaction requires some sort of conversation. In many of today's adventure games, these conversations are very controlled and predetermined by the writers and programmers of the game. Generally, the conversations consist of one fixed set of lines that the players can say to the particular character, whom the player has just met.

The purpose of our work is to create an alternative language interface to a character in an existing adventure game called "Den längsta resan". The language interface is alternative in the way that it is different from most of today's game dialogues, which are often very static, inflexible and completely predetermined. Our aim is to create an interface, which encourages the user of the game, i.e. the player, to become more interactive in its encounters with different characters in the game. By allowing the player to take control of the conversations, the game will become more interesting and fun and the level of difficulty will increase. Our aim is to combine linguistic knowledge and previous research on similar systems, such as agents, in order to achieve this alternative language interface.

In the game we have chosen to work with, which we also think is representative for most other adventure games, a menu of possible chat lines appears, when the player's character approaches another game character. These are the only lines that the player can use to interact with the specific character. Since the player does not have to think about what to ask the character, the game becomes controlled and the level of difficulty decreases. When playing an adventure game of this kind, the feeling of being on the outside looking in is very obvious. Most of the time it feels more like watching a film, than playing an interactive game, since every comment you can make has been determined before hand. It would be more challenging, if the player had to think of her own questions to ask each character in the game, instead of just choosing predetermined lines from a dialogue menu.

It is not only the player's lines that are static and inflexible. The other game characters, with whom the player converses, also have static lines ready for each of the main character's questions. Since the game characters always answer the same thing, when asked one particular question, they often appear dull and predictable.

Our idea is that the player of the adventure game is given the opportunity to formulate its own questions or comments in the conversations with other game characters. This will increase the interaction and give the player of the game a feeling of being in control. The game character on the other hand must, when asked a question formulated by the player, judge whether or not it has knowledge about what the player has just asked. If the character knows something about what the player is asking about, a reply will be generated, containing the requested information. By adding text generation, we can avoid making the game character seem static and predictable, since it can give differently formulated answers, when asked the same question repeatedly. Varying the output from the game character means adding randomness, which is also a goal in our work.

We will start this thesis off with a review of related work, which we have used as a starting point for our own work. We will then in Chapter 3 outline our own work and in Chapter 4 go into detail about the implementation of our system. Later, we will test our system and describe and also discuss the results of our work.

2 Scientific Background

The first kind of programs striving to communicate with users in a way similar to our interface was so called agents – programs designed to autonomously perform different kinds of tasks on the behalf of its user. The game character interface that we are designing might have the characteristics of an agent. Therefore, we will start by exploring what an agent actually is, to see if what we are trying to achieve is the construction of an agent. We will also describe some of the earlier work on agents and show some of their capabilities and limitations. We will also account for other relevant non-agent projects that have influenced us in our own work. Later in this chapter, we will describe many other linguistically relevant research contributions, concerning dialogues, knowledge, topic and text generation.

2.1 Agents

The idea of a system that is able to autonomously carry out different computer operations and tasks, in order to attain a given goal, originated with John McCarthy in the mid-1950's (Bradshaw 1997). The term 'agent' was coined by Oliver G. Selfridge a few years later, when he and McCarthy were both at Massachusetts Institute of Technology. Research on such systems has since taken many different directions, and the term 'agent' has been used irregularly to describe very different systems, ranging from scheduling programs and information retrieval systems to e-mail filtering programs. As agents of various kinds have arisen in different fields of research, such as artificial intelligence, human-computer interaction and end-user programming, there has been an increase in the use of the term 'agent', without a corresponding agreement as to what the term actually means. The term itself has become ambiguous and vague and there is a great need for clarification and definition of the term.

According to Bradshaw (1997) there are two different definitions of an agent that have been attempted. The first definition is based on the agent's ascriptive characteristics. Bradshaw points out that the word 'agent' comes from the Latin verb 'agere', which means to act, do, drive or lead. Therefore, the definition of a software agent must be that it should be able to act on the user's behalf and carry out particular tasks without constant user input. The agent must understand what the user wants it to do, by having knowledge about the context of the user's request. To do this, the agent must first also have knowledge about the user himself.

The second definition of the term 'agent' is based on the agent's descriptive characteristics. Bradshaw quotes Shoham (1997) and gives the following definition: a software entity that functions continuously and autonomously, in a particular environment, often inhabited by other agents and processes. Bradshaw emphasises

the importance of the qualities continuity and autonomy, since these qualities are essential for the agent to cope with flexible and dynamic environments. Ideally, the agent should also learn from its environment and adapt its own behaviour to the changes in its surroundings. According to Shoham's definition, an agent should also be able to co-operate and communicate with other agents, with whom they share a particular environment. Bradshaw comments on the fact that very few of the present agents, if any at all, comply with the descriptive definition of what is a true agent.

According to Bradshaw, in spite of the difficulties to fulfil the above-mentioned exacting definition, agents must nevertheless possess at least a few of the following attributes: *reactivity*, *autonomy*, *communication ability*, *inferential capability*, *temporal continuity*, *personality*, *adaptivity*, and finally *mobility*.

2.2 Earlier Projects

2.2.1 The Persona Project at Microsoft

This project (Ball et al. 1997) started with a vision of a change in the user interface metaphor: computers will become assistants rather than just tools. Around the world, computer users are already experiencing work and information overload, which is why there is a need to minimise user input and to increase the autonomy of systems and softwares. If software can perform large tasks autonomously, hiding as many of the details of the actual operation from the user as possible, the user workload will eventually be reduced. This would require a step away from today's direct manipulation interfaces, where the user controls every little operation by invoking a sequence of commands, which cause the software to carry out the intended task. Instead, the user will specify the overall goals of the task, and then delegate the responsibility for working out the details to the software. Ideally, in the specification of the overall goals, the user will need to describe the tasks to the computer, rather than just selecting them from a predefined list of alternatives. If the software does not understand its user's instructions, it should be able to ask the user for further clarification. The software should also, like a human assistant, be able to initiate actions on its own, when that seems appropriate in order to fulfil the user's goals.

So how will the specification process actually work? How can the user interact with the software and tell it what the goals are? In the Persona project, researchers realised that the most natural and convenient way of interaction was by spoken natural language. The belief at Microsoft was that sufficient progress on speech recognition and natural language understanding had already been made, so the prospect of a useful conversational interface seemed to be a realistic goal.

The Persona system is based on one character, the parrot Peedy, who operates a CD database and interacts with users, who want Peedy to give them information about artists, songs or albums. Peedy can search the database and answer a limited amount

of questions about the CDs and he can also play songs from the CDs in the database. Peedy responds to every given input from the user, with both visual and audio output. The Persona system is divided into three different subsystems:

- Spoken language processing subsystem
- Dialogue management subsystem
- Video/audio output subsystem

For the future work with the implementation of our interface, two of these subsystems will be of little interest, since we will not include speech recognition¹, visual animation or audio output in our system. On the other hand, the natural language understanding of the dialogue management subsystem is of great interest and that is why we have included the Persona project in this chapter. Each subsystem has its own goals and the Persona project's eventual goal for the spoken language subsystem is to allow users to express requests in natural conversational English, without having to learn a specialised command language. This is also something we want to pick up on, in the implementation of the language interface to our game character.

The dialogue management component functions as Peedy's brain, since it decides how to respond to perceptual stimuli. To enhance the believability of the character, the behaviour of Peedy is affected by memories of earlier interactions within the dialogue, or in previous conversations and by a simple model of its emotional state. The dialogue controller in the current system only covers sequences for a few conversational interactions, with no facility for managing sub dialogues.

The implementation of Peedy has mainly been coded in G (language transformation rules), but also in C, C++ and Visual Basic. The Persona project wants to point out that the prototype of Peedy is quite shallow in its capabilities, but that it is very effective at producing the illusion of conversational interaction. Despite the fact that Peedy actually understands very little, the user gets the impression that the agent understands the user input and is able to carry out a normal conversation, which is also the goal for the project. This illusion is also something we want to achieve in our own project.

Since one key goal in the Persona project has been to allow user flexibility to express their requests in their own words, in the syntactic form they find most natural, the interface is based on a broad-coverage natural language processing system, even though Peedy currently understands requests in a very limited domain. It is precisely the flexibility of spoken language that makes it such an attractive interface: users decide what they wish to say and express it in whatever fashion they find most

¹ If we were to actually intergrate our system into an existing game, speech recognition and analysis would definetly be considered.

natural. As long as the meaning of the statement is within the range of Peedy's understanding, he will respond in a correct manner and carry out the user's request.

The Persona project system constructs a rich semantic representation of each user utterance, which is then mapped into a task-based semantic structure. In the current Peedy prototype, all possible utterances are described to the system by a context free grammar.

2.2.2 *Eliza*

The Eliza program, named after Shaw's Eliza Doolittle, was created in 1966 by Joseph Weizenbaum (Hutchison 1986). It consists of two separate modules: a language analyser and a reply generator. The generator will, for one specific domain, create an English sentence. The most famous domain is where Eliza appears as a psychotherapist. The early users were often very surprised by the system, and some meant that it demonstrated a general solution to the problem of making computers understand natural language. Many of the users actually thought that Eliza really understood them, which could imply that Eliza passed the famous Turing test².

Some human psychotherapists even believed that Eliza could be a future co-worker, since she could help so many patients at one time. But Eliza was in no way an understanding psychotherapist. One of the reasons for the illusion that fooled so many of the early users, was Eliza's ability to connect an input word with a series of hypernyms³. If the user for example says something about his mother, Eliza can then ask him to tell her more about his family. The system uses sentence patterns and keywords to analyse the input, and the answers are more like imitations than human responses.

Let us look at an example.

user: *everyone is crazy except you and me*

Eliza first matches the input string to sentence patterns. The matched pattern looks like this:

<words> you <words> me <rest>

² The Turing test, created by mathematician Alan Turing, aims to determine whether a system is intelligent or not. In the test, a judge has conversations (via teletype) with two other participants; one human and one machine. If, at end of the conversation, the judge cannot distinguish the machine from the human, the machine is said to be intelligent.

³ A hypernym is a word whose meaning denotes a superordinate. For example, one of the hypernyms for the word 'robin' is 'bird', another is 'animal' etc.

It is assumed that the two words 'you' and 'me' are separated by a verb, which surprisingly often seems to be the case. Eliza will therefore change 'you' to 'I' and change 'me' to 'you' and then quote the words between the two pronouns. In this case, where the word in between actually was not a verb, the result will take the shape of a completely absurd reply.

eliza: do you think you and I?

Looking at Eliza today (see WWW3), it is amazing that she ever actually fooled anyone in those days. The positive reactions from the users imply that they must think of themselves as good conversation partners, since Eliza mostly imitates their input. Eliza though shows us that relating words to their hypernyms is one way to create an image of understanding and human behaviour.

2.2.3 The August Spoken Dialogue System

August was developed at KTH in Stockholm and was available for the public during the Culture Capital of Europe-program (Gustafson et al., 1998). The system has a speech interface and two screens: one screen which has an animated face, looking like the late Swedish author August Strindberg, and one screen that displays additional information. This information is August's knowledge, consisting of information about restaurants, maps of Stockholm, the Yellow Pages, the August system itself and about the author Strindberg. These different areas of knowledge are stored in separate files, which means that the analyser of August first has to assign an area to each given input. This is done by searching for keywords, like 'map', 'square', 'Hemsöborna' etc. The separated files make up for a clearer structure of the knowledge and this kind of storage makes it possible for other agents or systems to use one or more of the information files.

August has a simple dialogue manager, and the developers made an important observation: if August doesn't reply to the user's input within a few seconds, the user assumes that the input wasn't understood by the system, and the same question is often repeated or rephrased. This behaviour often leads to a strange dialogue, since August then receives a queue of messages and answers them one at a time.

2.2.4 Julia

Julia is a MUD (Multi-User Dungeon) agent, developed at MIT Media Lab, which has the ability to interact with other players on a MUD⁴. Julia behaves like any of the other players on the MUD, but is also capable of answering questions about the

⁴ A MUD is text-based multi-user simulation environment, not too different from a chat channel, in the way that people interact with each other using written natural language (Foner 1997).

different rooms or players. In this sense, Julia can therefore be used as a guide on the MUD. Julia has a simple discourse model, often sufficient enough for the simple structured discussions in which she participates, though sometimes the simplicity of her discourse model means that she ‘forgets’ what she has already asked or been told, and therefore repeats herself.

Julia has an ego, which means she can talk about herself, and answer questions about her mood or actions. She can guess the sex of a player from his or her name, and come up with nicknames by using the name’s first letters. The input from the users is analysed by matching it to patterns. Each pattern is connected to certain answers. Most of Julia’s replies are hard-wired, which means that there is little flexibility in the question-answering structure.

However, since each pattern has a large number of possible replies, Julia seems to answer the same questions differently each time that she is faced with it. Users are therefore often quite impressed with Julia’s human-like language usage, at least initially. Some people do not even figure out that she is a MUD agent, but evaluations of Julia show that most people after a while realise that she is not human. What often gives her away, is her limited dialogue manager, which leads her to repeatedly ask the same questions to the same players. When she does not understand (i.e. when she does not recognise) a phrase, she asks for a rephrase or starts a new discussion by completely changing the subject, sometimes in form of a cliché. Her favourite subject is ice hockey, which she often wants to talk about and also starts discussions on.

Julia is impressive, and the most human-like agent we have tested or read about. Transcripts from some of her interactions show us a wide variety of possible replies, and as mentioned above, it is not obvious for all the other players that she actually is a machine. Julia has the ability of referring to things or people earlier discussed, to handle anaphors and to talk about her own feelings. In some discussions these abilities are taken for human and she has managed to pass the Turing test on occasion. Things that reveal her secret of being an agent are the speed of her answers (she answers a lot quicker than most people can type) and her bad memory and repetitions of earlier facts and questions. She also lacks some common knowledge, for example that she does not know what the Stanley Cup is, even though her favourite area of discussion is ice hockey. A suspecting player can also ask questions about the implementation of Julia, and she will give away her secret, by answering with exact figures on memory capacity and library addresses for the source code.

2.3 Input Analysis Theory

2.3.1 Identifying Topic

To understand a human's contributions in a conversation, whether it is a MUD player chatting to Julia or a player interacting with a game character in an adventure game, there must be some way of analysing the meaning of each contribution in the conversation. So what is a contribution about? Linguists have for a long time been occupied with this question which, although it is easily formulated, has proven difficult to answer: What is an utterance about? The difficulties can be explained by looking at a very simple example. Take the sentence "John loves apples". Is this sentence about John, love or apples? Without context or intonation, it is impossible to say what one isolated sentence is about.

Peregrin (1995) therefore gives a more syntactic point of view on what an utterance is about. In a neutral context and with a neutral intonation, he says, the utterance is about John. In this sense, the topic of an utterance has to do with the structure, or basically the subject. Allwood and Andersson (1996) mean that this most often (though not always) is the case: subject, agent and topic are all basically the same and appear as the first clause of an utterance. Allwood and Andersson further make the distinction between topic and comment by saying that topic is the old information, and comment is the new, added information of the utterance.

Peregrin has another definition of topic. The sentence "John walks" could be the answer to the question "What does John do?". According to Allwood and Andersson, the old information first appears in the question, and the topic would be 'John'. Peregrin means that since the question is addressing John, 'John' is the semantic subject, which is also his definition of topic. From these viewpoints, it is now independently proven that if the question had been "Who walks?", the topic would have been 'walks'. However, from these two definitions, it is still impossible to define the topic of an isolated utterance, without the context.

Levinson (1993) does not give a proper linguistic definition of the topic of an utterance, although he analyses the way a conversational topic is introduced, closed and changed within a discourse. After the opening utterances of a conversation (such as greetings and recognition of speakers), the topic, or the cause of the conversation, is often raised by one of the speakers. (Levinson mainly discusses telephone discourse, where the calling participant mainly raises the topic.) The closing of a topic is often recognised through affirming statements, such as 'okay', 'that's fine' or 'let's do it'. The prior discussion is now seen as closed, and a new topic can be expected. But a new topic can be introduced without the prior one being closed. This change of topic is often, though not always, marked with a phrase, such as 'hey', 'about something else' or 'by the way'. Levinson therefore gives us some hints when to expect a new topic, and when not to, that is when anaphors still refer to a prior topic.

Not much work has been done in identifying topic within isolated sentences. If this is because that task is not even solvable, or because no one has tried to solve it yet, we cannot say for sure. As far as we can tell, there is yet no communicative agent with the ability to identify the topic of an isolated sentence. The major research interest on topic nowadays seems to be in text summarisation, where identification of topic takes place on the basis of a whole discourse or text.

2.3.2 Formalising the Intentions and Plan of a Dialogue

A good conversation partner knows what question to answer, even if it is not explicitly uttered. This is not only the case with implicit questions, such as “Do you know what time it is?”, but also when it comes to being helpful and answer explicit questions with a direct answer, with some further information. Allen and Perrault (1980) declare that these are two good ways of making a question answering system helpful. They claim that discourse (at least in a question answering system) is about the participants’ goals, and the questions that arise in the dialogue are parts of the plan to reach this goal. More formally, they define a plan in the following way: *Given an initial world state W and a goal state G, a plan is a sequence of actions that transforms W into G.* (Allen and Perrault, 1980). It is essential for both speaker and hearer, at one point or another in the dialogue, to become aware of each other’s plans.

Allen and Perrault claim that many of the hearer’s helpful utterances are given when the hearer detects an obstacle for the speaker to reach his goal. If the speaker says “Do you know what time it is?” the hearer does not only detect the speaker’s lack of knowledge of the time, but also that knowing the time is necessary for the speaker to achieve his goal. Therefore, the hearer does not only answer the question with ‘Yes’, but also by telling the time (if he knows it). Formalising this sort of implicit utterances, they say: “...if X wants Y to want to do ACT, he may achieve this by getting Y to want to achieve E, where Y’s planning process will infer ACT as a way of achieving E.” (Allen and Perrault, 1980) This is what they call the Nested-Planning Rule.

One way for us to implement the human ability to give (more) information to an implicit question, could be to always let the system answer a yes/no- question with either just “No”, or “Yes” accompanied by a further question or statement related to the topic. It would seem like the system is being helpful and that it knows that implicit questions like “Do you know what time it is?” not only wants the explicit answer, but some further information.

2.4 Knowledge Representation

Knowledge is something quite abstract. One cannot physically see or touch it, though it is possible to talk about it and also, in some more or less mysterious way, to store it within a memory to use it again later. We also have the ability to process our knowledge, i.e. we can use some of the stored information to retrieve new knowledge. These are the functions that an information based system has to deal with: storing and processing.

In a computational system, the storage of knowledge is called the knowledge base (KB), and the functions for processing the knowledge make up the inference machinery. There are many opinions on how to best represent knowledge, and what knowledge actually is. First order logic, FOL, is a language trying to formalise meaning, though not sufficient enough for some propositions. The difficulty of representing semantics has used many approaches, and most of them include some metalanguage, describing semantic structures using hierarchies or frames, with various success. There is, however, no all-success-formula.

Lager (1990) stresses the importance, in analysing text, of keeping the semantic representation open, even ambiguous, enough for the sake of possible anaphoric resolution. He exemplifies this by comparing two text analysis systems, which both use a kind of FOL. The text “A dog chases a cat. It barks.” would be interpreted as (1) and (2) by these systems.

- (1) [barks(X1), cat(X2), chase(X1,X2), dog(X1)]
- (2) [barks(X3), cat(X2), chase(X1,X2), dog(X1)]

In (2), the variable X3 is stated to refer either to X1 (the dog) or X2 (the cat). If a user now asks these systems “Why does the cat bark?”, they will give different answers. The first system would probably reply with something like “The cat does not bark at all”. The second system has the possibility to now resolve the anaphor, meaning that “it” could actually refer to the cat, since the system itself has not yet decided on this. Instead, it keeps the semantic interpretations open for all possibilities.

Brachman et al. (1983), propose another approach to knowledge representation, not using framebased languages, in the creation of KRYPTON. This system uses two different languages; one for storing knowledge and another for processing it. The reason for not using frames, as they explain, is that semantic frames tend to be ambiguous, leaving too much of the interpretation to the user.

So what we have here are actually two contradictory opinions. Brachman et al. do not use semantic frames because of their ambiguity, and Lager who claims that an analysis system should not itself resolve anaphors, but keep the ambiguity open for different interpretations.

According to Danlos (1997), most semantic representations presuppose the existence of a language free meaning representation, which is common for all natural languages. He argues that this is not consistent with Whorf's studies on how speakers of different languages do not perceive the world in the same way. He also describes problems raised using natural language representing knowledge. He points at the difficulties of connecting related words to each other, showing the example of 'buying' and 'selling'. With a semantic representation, it is possible to say that if P sells X to Y, this means the same as if Y buys X from P. He also claims that that when storing knowledge using words in a specific language, it is hard to vary the generated sentences or to use the representation for producing text in different languages.

It seems to us, that there is no way one can represent knowledge in an all-successful way. Every different method has its own set of advantages and drawbacks.

2.5 Text Generation

In earlier research on text generation, most work is done on generation of whole texts, not on isolated sentences, or in the field of text summarisation. SUMMARIST (Hovy and Lin, 1997) is a system for summarising one text. TEXT (McKeown 1985) generates text from semantic representations within a query-answering system and Stede (1999) uses a semantic representation to generate multilingual texts, i.e. translations in English and German. The Compansion Project (Demasco and McCoy, 1992) generates single sentences out of content words.

2.5.1 TEXT

TEXT (McKeown 1985) is a generation system based on a database of relations between military devices. The user can ask about the information available in the database, the definition of a term or the difference between two terms. The system uses discourse strategies and information on which word is currently in focus to know what to generate next. TEXT generates sentences in passive and active voice and the use of anaphora and there-insertion is also implemented. No input analysis is performed, except for mapping the input to the three possible questions: 1) What is X? 2) What do you know about X? and 3) What is the difference between X and Y?

When receiving a question, TEXT chooses a schema on how to structure the answer. The content of the question will be the source of which semantic representation is retrieved from the database. This representation is passed on to a dictionary interface, translating the semantic representation of propositions, one at the time, to a deep structure representation of the entire sentence to be generated. A functional grammar

then makes an English sentence out of the deep structure, consulting the schema to fit the sentence into the previous context.

If the input question for example is “What is a Hobie Cat?”, TEXT realises the insufficiency in the direct answer “A Hobie Cat is a type of catamaran”, using the hierarchical relation of the database. The focus is now on ‘catamaran’ and TEXT therefore generates an explanation on what a catamaran is, perhaps by using the relevant knowledge and explaining the difference between an ordinary sailing boat and a catamaran. The system uses no user model, but treats all users as naive, presenting as much information as possible on every input question.

Creating the TEXT system dictionary is a time consuming work. For each entry the designer has to consider the following: 1) Its possible functions in a sentence, 2) What syntactic categories should respond to each of its functions, and 3) What lexical items should be used within its functions.

Phrases generated by TEXT can be conjoined, either by ‘and’ or by ‘or’. Prepositional phrases often uses ‘or’ as in “in the air or on the ground” and nominal phrases uses ‘and’. No efforts have been put on the sentence level, where many other conjoining terms, as ‘but’ and ‘although’, also might be necessary.

TEXT treats compound English nouns, such as ‘fuel type’, as one unit. In the lexicon, the words that make up the compound word are concatenated with ‘_’ and stored as ‘fuel_type’, and put to the user in the same way.

2.5.2 The Compansion Project

The Compansion⁵ Project at the University of Delaware has developed a system for creating a full grammatical sentence out of a list of content words given by a user (Demasco and McCoy, 1992). The system is mainly intended to be used by physically disabled or people with a diminished ability to produce written language. The basic idea is that the user should only have to type the content words of the intended sentence, and the system will then generate all the formal possible sentences, of which the user can choose the one he intended. To do this, the system uses three different modules.

The user’s input is a content word sequence. Using this list of words as input, means that the system can also cope with ill-formed input (McCoy et al., 1989). The sequence is first processed by a semantic parser. A sequence like [john, hit, hammer, mary] would mean that each word is given features and a possible function in the output sentence. ‘john’ will probably be assigned features like ‘noun’, ‘animate’,

⁵ The term *compansion* is derived from the two words *compression* and *expansion* (Demasco et al., 1989).

‘human’ and ‘male’. The possible functions for ‘john’ are either agent or beneficiary of the action. Most important for the parser is to find the main verb of the sentence, and also the complements that this verb demands. Some verbs do not require complements and some cannot take them at all and all this information must be recorded in the verb features in the lexicon.

After identifying the verb, it is possible to define all of the syntactic functions of the content words. All words are classified into one or more roles: verbs, objects, adjectives, adverbs and prepositions⁶ (McCoy et al., 1990). In the earlier example, both ‘john’, ‘mary’ and ‘hammer’ could be possible agents, though it is more likely that ‘hammer’ is a tool to the main verb ‘hit’, and since there are no other candidates for this function, ‘hammer’ takes that role. Both ‘mary’ and ‘john’ are equal candidates for the agent role, and both of them are also equal candidates for the one who is exposed to the hitting. The semantic parser leaves this decision to another module. The bottom-up parser is accompanied by a top-down one, starting at sentence level searching among the content words for possible constituents. If no verb is mentioned in the input sequence, the system adds a form of one of the verbs ‘be’ or ‘have’, depending on what kind of verb the other words demand. If no possible agent occurs in the sequence, the system assumes that it is about the user, and adds ‘I’.

This information is then given as input to the next module, which is a representation translator. The content words will here be arranged in the correct word order, which implies that all the syntactic functions must first be decided. In a case like this, where no distinction can be made between ‘john’ and ‘mary’, no decision is made. The translator also tries to define the focus of the input, by checking the previous input, the context. The sequence produced by the translator has decided the word order of the grammatical sentence that is to be generated.

As input to the generator, there are now several word order sequences from the original content words. The generator will conjugate subject with main verb and give the final possible grammatical output, containing the content words and additional prepositions and particles. The possible outputs of our earlier sequence would contain “John hit Mary with a hammer”, “John was hit by a hammer by Mary”, “Mary hit John with a hammer” and “Mary was hit with a hammer by John”. The generation is based on a functional unification grammar.

This system was developed in the late 80’s and early 90’s. The lexicon was at first very small, since every word manually had to be assigned the information of features and possible syntactic functions. A module for unknown input word was later developed (Jones et al., 1991). Starting with the definition of the main verb, the parser then realises what function is missing and assigns the unknown word to this

⁶ The input is supposed to be given by the user, which means that not only content words can be expected.

role. The unknown word is then put into the lexicon, but it will not get any detailed information about features and function, since the use is just an assumption.

2.5.3 SUMMARIST

SUMMARIST is a text summarisation system, based on the equation *summarisation* = *topic identification* + *interpretation* + *generation* (Hovy and Lin, 1999). The system is clearly developed based on these three components. Hovy and Lin discuss the difference between an *extract* and an *abstract*, in that the first quotes important sentences of a text, whereas the second interprets and rephrases the important content. SUMMARIST is designed to do both.

The topic identification component of SUMMARIST is the most refined one. It uses multiple techniques for identifying topics of a text, e.g. the words' positions in a text (titles and headlines), word count and cue phrases, such as "in conclusion" and "the most important". If the output from SUMMARIST is to be an extract, this component is the only one needed.

After extracting the words considered to be the topics of the text, these topics have to be interpreted. This is, according to Hovy and Lin, the most difficult process, since information that often not is present in the text has to be known. For example, if "He bought milk, apples, bananas and meat" appears in a text, this could be interpreted and shortened to "He bought some groceries" For this, SUMMARIST uses WordNet's world knowledge, making it possible to generalise the more specific terms to a hypernym. To be able to understand that the topics 'enter', 'wait', 'eat', 'pay' and 'leave' could be interpreted as a restaurant visit, SUMMARIST has built a conceptualiser, clustering words that appear in the same domain.

The generation module of SUMMARIST is not yet fully implemented (Hovy and Lin, 1999), meaning that what you actually now can use SUMMARIST for, is getting an extract or a topic list with the interpreted topics. The intention is to produce more readable abstracts, adding phrase concatenation and full sentence planning and generation.

2.5.4 On Verbs and Prepositions

There is a problem in generating sentences that only consist of grammatically correct verb phrases where the verb is followed by only correct phrases (subcategorisation) and/or the correct prepositions. Dougherty (1994) presents a possible solution in a grammar where he uses the Prolog predicate `append/3` to build sentences out of phrases, as in (1).

(1) `s(X) :- np(A), vp(B), append(A,B,X).`

Using `append`, he also makes restrictions on what preposition goes to what verb, (see 2). The last of the two calls of `append/3` in this rule for building verb phrases, sees to it that the only word immediately following the verb, has to be the correct preposition.

$$(2) \text{ vp}(X) :- \text{v}(A,\text{Prep}), \text{pp}(B), \text{append}(A,B,X), \\ \text{append}(\text{Prep},_,B).$$
$$\text{v}([\text{depend}], [\text{on}]). \\ \text{v}([\text{look}], [\text{at}]).$$

For verbs that can take two preposition phrases, as ‘go’ that can take both ‘from’ and ‘to’, the grammar rules that apply forces ‘go’ and its prepositions to be listed twice in the lexicon, since it is correct to say both ‘go from something to something’ and ‘go to something from something’ with the prepositions in different order.

For the lexical entries in the lexicon, Dougherty does not only separate transitive verbs from intransitive, but also transitive verbs that take noun phrases as complement from those who take prepositional phrases. This is, as he mentions, not a strictly linguistic categorisation, since in a linguistic point of view, all transitive verbs are transitive in the same way, meaning that they take a complement of any kind. His division implies that a verb that can take either a noun phrase or a prepositional phrase as complement has to have two different listings in the lexicon. This increases the size of the grammar, since all different verb types have to have their own grammar rule(s), but it also decreases the number of incorrectly generated verb phrases.

The simplicity of Dougherty’s (1994) grammar is striking, and in fact, it is a bit too simple. The words in the lexicon have no features and even though the grammar would make a good ground for parsing, the lack of features makes it a lousy basis for generation. In spite of this, Dougherty has shown us a way to solve the verb phrase complement problem, using the `append` predicate.

2.5.5 Language Generation Tasks

Dale and Reiter (1995) describe the tasks that must be performed by a language generation system, mainly focusing on text generation systems. They first enumerate the different tasks that the natural language generation systems perform and then discuss possible architectures and representations for supporting these tasks. They see the task of a natural language generation system as a form of mapping of some input data to an output text. (This, however, will not be exactly like our view of NLG systems, since we will take input from a user, analyse that input and then generate output back to the user, in a form such that the user feels that the output is a

believable response to the input sentence.) Even though we might not wholly apply Dale and Reiter's view of NLG systems to our system, we will describe their basic idea, since there are some things we can pick up from their work.

Dale and Reiter mean that it is always necessary to decompose computational processes into a number of more finely characterised substeps and they also do this in the task of natural language generation. There are six basic kinds of activity that need to be carried out in going all the way from input data to a final output text.

The first activity is content determination, which is the process of deciding what information should be communicated in the output text. Dale and Reiter describe this process as one of creating a set of messages from the system's input. These messages are then used in the language generation process that follows. Generally, the message creation process largely consists of filtering and summarising input data, and the messages are expressed in some formal language.

The second activity is discourse planning, which is described as the process of imposing ordering and structure over the set of messages to be conveyed. Note that this use of the term 'discourse planning' is very different from other researchers' view. The next activity is the sentence aggregation, which actually groups messages together into sentences. After that follow lexicalisation, which is the process of deciding what specific words and phrases to use in expressing the concepts and relations, which appear in the messages.

The fifth activity is the referring expression generation, which is the task of selecting words or phrases to identify domain entities, such as the use of anaphors. The last activity is the linguistic realisation, which is the process of applying the rules of grammar to produce a text that is syntactically and morphologically correct.

2.6 Discourse

In order to model human-human dialogues in adventure games, we need to include some of the important aspects of such dialogues. In this section, we will look at some of these aspects, such as how humans take turns in conversation and how two participants collaborate with each other when engaged in a dialogue. Our aim is to include some of these aspects later on in our implementation of the natural language interface.

2.6.1 Discourse Structure and Turn-taking

Levinson (1993) defines discourse as a set of utterances, often paired in two, following each other in a certain pattern. These adjacency pairs consist of one

opening utterance and one closing one, uttered by two different persons. The pairs are categorised so that one type of opening demands the same type of closing, for example does the opening utterance of type greeting, demand that the other person utter a closing utterance of the same type, greeting. The pairs do not have to be directly following each other, but they can be seen as opening and closing parentheses, and can be nested. This is exemplified by the following conversation:

greeting_open	A: <i>Hi!</i>
greeting_close	B: <i>Hello.</i>
question1_open	B: <i>Do you want an ice cream?</i>
question2_open	A: <i>What flavour is it?</i>
question2_close	B: <i>Chocolate.</i>
question1_close	A: <i>Yes, please.</i>

The pattern of this conversation can be formalised as (go, gc, (qo_1(qo_2, qc_2), qc_1)).

These analyses and structures are best applied to conversation between two participants. If there are more people involved, the structure gets more complicated, for example since different topics can be raised at the same time, and anaphors therefore can refer to different topics, sub conversations can start without greeting utterances or end without a closing and there is no obvious turn-taking.

2.6.2 Discourse, Collaboration and Mixed Initiative

The following research on discourse is interesting for our input analysis, since it describes many aspects and characteristics of human conversation. Since we want to make the interaction between the player and the game character as flexible as possible, we need to take a deeper look into human discourse patterns and later apply this knowledge on to the human-computer conversation. Therefore, we must first find out what features are significant to human conversation. Clark's research on discourse and collaboration (1992) has mainly been done on verbal conversations between people, but we feel that the same results would apply on written conversations, such as chats, dialogue systems or question-answering systems.

Clark (1992) has looked into how speaking and understanding work in conversation. He stresses that this question should be central in psychology, but since psychologists have failed to study this field, he draws upon research done in sociology and philosophy to answer this question. Clark agrees with Walker and Whittaker (1990) that conversation is a collaborative process between two or more participants. In conversation, the participants work together in many ways, for example in the making of a definite reference. In Clark's model, the speaker initiates the process by presenting a noun phrase. Before moving on in the conversation to the next contribution, there might be some repair or expansion on the noun phrase, until

the participants reach a mutually acceptable version of what the noun phrase refers to. Clark argues that his proposed model follows from the mutual responsibility that participants in conversation bear toward the understanding of each utterance.

Clark has explored one essential use of language: how people co-ordinate their efforts in conversation to make a definite reference. Clark has not concerned himself with semantic reference, but with speaker reference. The difference between these two types of reference lies in the following example: the phrase “the cat with the long tail” has the semantic reference, translated into first order logic:

$$\exists x(\text{cat}(x) \ \& \ \exists y(\text{tail}(y) \ \& \ \text{has}(x,y) \ \& \ \text{long}(y) \) \) .$$

The speaker’s reference, when uttering the phrase, is the actual cat the speaker is talking about, perhaps as part of an assertion he or she is making about owning this cat.

Clark claims that making such a reference is a collaborative process, requiring actions by both the speaker S and the listener L. When S uses the noun phrase “the cat with the long tail”, he intends for L to uniquely identify one specific cat, possibly from a group of many different cats. Using the noun phrase, S satisfies his intentions by making the noun phrase an issue or topic. The listener L hears the definite description and hopefully infers the identity of the referent.

Clark sets up the following literary model from this example:

- 1 The reference is expressed linguistically with one of three standard types of noun phrase: proper noun, definite description (det + n), or pronoun.
- 2 The speaker uses the noun phrase so that the listener will be able to identify the referent uniquely against their common ground.
- 3 The speaker satisfies his intention simply by issuing of that noun phrase.
- 4 Only the speaker controls the course of the process.

The idea behind Clark’s model of reference is that the participants of the conversation must mutually accept that both of them understand what the topic of the conversation is. To do this, they must therefore establish the mutual belief that the listener L has understood what the speaker S has just said, before the conversation can proceed. Clark argues that establishing this mutual belief of understanding takes the following acceptance process: (1) a presentation, and (2) an acceptance.

S makes the presentation “the cat with the long tail” and in doing so believes that L from the description can identify the referent of the noun phrase. Once S has made the presentation, L must accept it. There are two methods for acceptance. First, the listener can presuppose acceptance, by continuing to the next contribution or by allowing the speaker to continue. Letting the next contribution begin is, according to Clark, equal to an acceptance of the last presentation. The second method of acceptance is to assert the acceptance, as with continuers such as ‘yes’, ‘right’, or ‘I

see'. These continuers are also mutually recognised as acceptances of the last presentation.

If the listener does not want to accept the speaker's last contribution, he must communicate this to the speaker. There can be several reasons for why a listener does not immediately accept a speaker's presentation. The listener might not accept the description of the referent, and respond with "Those aren't real cats, they are just toys." or he might not see the description as sufficient enough to uniquely distinguish the referent and respond with "They all have long tails." When the listener does not accept the speaker's contribution, and has communicated this to the speaker, the speaker must deal with the listener's implicit or explicit questions until the listener finally accepts the presentation. This process is evidence that participants in a conversation are mutually responsible for establishing what the speaker means, and definite reference is only one part of that process.

While the collaborative process in a dialogue goes on, another characteristic of human conversation becomes relevant and that is that humans in dialogue with each other take initiative in the conversation. Taking initiative means taking control over a segment of the discourse.

When two people take part in a conversation normally there is mixed initiative between the participants and both of them share control over the conversation. Walker and Whittaker (1990) apply a set of rules for the transfer of control in dialogues. The application of these control rules derives domain-specific discourse structures, which indicate that initiative plays a role in the structuring of discourse. The analysis performed by Walker and Whittaker suggests that discourse participants often mutually agree to a change of topic.

Since conversation is a collaborative process, models of conversation can provide the basis for extending planning theories. When the situation requires the negotiation of a collaborative plan, these theories must account for the interaction beliefs of the two participants. According to Walker and Whittaker (1990), previous attempts to imitate human-to-human discourse in human-computer interaction have failed partly because of the lack of mixed initiative. The mistake of ignoring this characteristic in dialogue systems, gives rise to two problems: the user cannot participate in the reasoning process or ask questions he or she might have and, in addition, question answering systems often fail to acknowledge the system's role as a conversational partner. For example, fragmentary utterances may be interpreted as referring to the previous user input, but what the user was actually saying was a response to the system's previous answer.

2.7 Summary

In this chapter, we have discussed various fields of research, including research on agents, knowledge, text generation and discourse, indicating the width of our work. We will rely on many of the discussed theories and we will also be influenced by many of the earlier projects, resulting in a natural language interface that makes it possible for the player of a game to interact with the other game characters, in a flexible and free manner.

The Persona Project resembles our own project in the way they chose to store information. However, they used a CD database, where we will use a knowledge base for storing information. The task is to retrieve and answer questions on this knowledge. The Persona Project chose to implement an agent and gave this agent a personality and a name; Peedy. The use of natural language is common to all of the discussed projects; Peedy, Eliza, August and Julia all converse in English. All of them match input words or sequences to hard-wired patterns, and respond with certain hard wired answers. The variety of output depends on how many answers the programmer chooses to implement.

As described earlier, to define one single sentence's topic is very difficult. However, we will use the theories that do exist in order to identify the topic of the player's questions. When it comes to sentence generation, we have described earlier work with different purposes. Most of the systems described in this chapter generate whole texts, only the Companions Project stays on the sentence level. It is only the Companions Project that does not represent knowledge with semantic structures. Instead, they include extended syntactical features in the lexicon. We have discussed some of the many theories on conversation and discourse both from a strictly theoretical and from a more practical point of view, showing some features applicable to a natural language interface.

3 Methods

3.1 Overview of the System

Our work consists of implementing the language interface to a game character in an adventure game, so that the player of the game can ask relevant questions to this game character. Agreeing with the Persona Project, that the most natural and convenient way of interaction is by natural language and not some sort of specialised command language, we want the player to be able to put these questions in her own words, in the syntactic form of her choice. This is one goal that we share with the Persona Project, to allow user flexibility enough to let the user freely formulate its own questions.

If we would choose to implement the game character interface as an agent, we would have to devote a lot of time and effort to create the different characteristics that make up an agent, as described above in Section 2.1. Our game character will have the agent quality of communication ability, since we will implement the ability to communicate with others using natural language, but it will be difficult for us to implement any of the other agent characteristics within the framework of this thesis. Even if it would be theoretically possible for us to implement some of the typical agent traits, such as personality and temporal continuity, we here make the choice of not implementing our game character as an agent, but as an interface.

We will here describe the game scenario of the chosen game and give the game character knowledge about a certain area of the game, to limit the boundaries for the KB and text generation. Our work can be divided into at least three different parts (as shown in Figure 1):

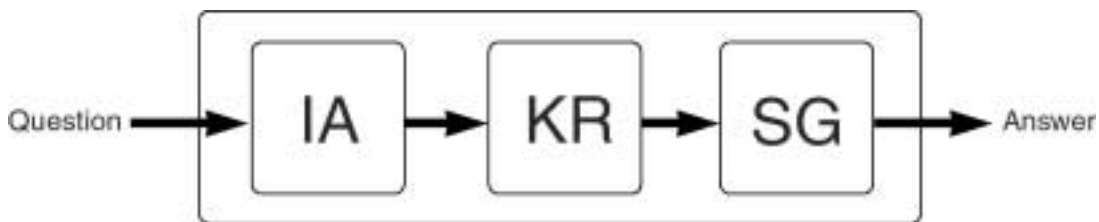


Figure 1: System overview showing input and output and the flow between the main modules (Input Analysis, Knowledge Representation and Sentence Generation).

1. Processing of the player's input questions.
2. Searching in a knowledge base, KB, and looking for possible answers to the player's questions.
3. Generation of replies to the player, where the demanded information is included.

The focus in the implementation of our interface will be on the output generation, where replies to the player's questions will be generated, using the knowledge in the knowledge base. We will use the content words in the knowledge base to form sentences, using a grammar and a lexicon, and also a generation method implemented in Prolog, all of which are written by us.

This thesis will mainly focus on the last part, i.e. the implementation of the generation module, but in order to achieve a whole functioning unit, we also need to implement the other parts of the system, although in a limited extent. This, for instance, means that the input analysis will not be able to actually understand what the player is saying, since that would require a full semantic analysis of the player's input, which would be too time consuming for us to implement in the scope of this thesis. Instead, we will search for content words in the player's input sentences and match these with the knowledge base, to see if our game character knows anything about these content words. We will however not do this in the hard-wired manner as seen in Julia or August, but we will use a number of methods when searching for a match between the topic of the player's question and the knowledge stored in the KB.

The knowledge base will also be limited in its implementation and we will tag each of the content words (which are the keys in the knowledge base) with a tag that tells us what kind of information we have about every word. In theory, we could use a large number of different tags, to capture exactly what kind of knowledge our game character has about each content word. We could use tags for *where* the thing is positioned in the game, *what* the thing is, what it can be *used for*, who *owns* the thing, *why* the player must find the thing and so on, but since we cannot devote too much time on this part of our work, we will only use a small number of tags and instead use one general tag that is used for 'unspecified' knowledge. This means that we will not have to analyse exactly what the player is asking for in one input sentence, but instead just find what content word is being questioned and if one of our chosen tags is relevant in the current situation. If there is no tag matching one specific question, the general tag will be used and the generation module will generate an answer containing unspecified information to the player on the topic in question.

In our knowledge base, we will only use the tags 'where' and 'who' and also have a general tag 'g'. If a player asks a question about one of the content words included in the knowledge base, for example a 'plate', but does not include one of the wh-words that form tags in the knowledge base, a general information about the plate will be generated, for example "The plate opens the tower." If the player instead had asked about the position of the plate, and included the *where* keyword in its input question, the 'where' tag would have been used, and the information generated might then be the following: "The plate is in Arcadia". Note that we do not differentiate between any other types of questions, apart from *who* and *where* questions, which means that

any other type of question will be interpreted using the general tag and unspecified knowledge will be generated.

Inspired by Eliza's structure, with separate modules for analysing input and generating output, we have divided our work into three separate parts: an input analysis module, a knowledge representation module (KR) and a sentence generation module. Our goal is to create this system from start to finish, taking natural language as input and giving natural language back as output. The two last modules, the KR and the generation, are closely related. We have decided to use English when representing knowledge in the KB, and to send this knowledge straight to the generation module, without any intermediate processing. We will also use a post-processing module, which is added after the sentence generation, to conjoin sentences and to make use of anaphors in sentences that share subjects. The input analysis will be as simple as possible, though efficient enough for our purpose.

We have also made the choice of not implementing an overall discourse module that keeps track of the shifts of control in the dialogue between the player and the game character. This is an important and well-grounded limitation in our work. It would have been very interesting to have made use of a dialogue manager, that could have allowed both the game character and the player to take initiative in the conversation. However, since our main goal has been to get a whole functioning system, from end to end, we realise that we do not have enough time to implement the finesse of a dialogue manager. However, we will later on in this thesis account for the effects of such a module, in a theoretical discussion (see Section 5.3).

3.2 Tools and Assets

In this section, we will describe all of the tools and assets that we have used in the implementation of our own system. We will start by describing the adventure game, including the original dialogue that we have used as a model for our work. We will later describe the external programs that we use in our system and we will round off with a description of the modules we ourselves will implement.

3.2.1 The Adventure Game

In this section, we will give details about the chosen adventure game, called "Den längsta resan"⁷. We will start by describing our idea on how we want to change adventure game dialogues and later describe the main story of the game. We will also show what the original game dialogue looked like.

⁷ The game now exists in an English version, called "The Longest Journey" and it is made by Funcom and distributed by IQ Media Nordic AB. See www.funcom.com for further details about the game.

The Idea

When we came up with the idea to change adventure game dialogues, to make them more interesting and flexible and less static and predictable, we already had a particular game in mind. We had recently played a game called “Den längsta resan” and found this game to be graphically very impressive. However, we felt that the game’s dialogues were its biggest drawback, since the dialogues are very controlled and static and do not involve the player’s own language capacity. When playing the game, we often felt bored during the long dialogues, since we, as players of the game, did not get to contribute any input, but became more like spectators of a film.

In the game, there are over 100 encounters with different game characters, with which the player’s character must talk, in order to retrieve information about how to solve the different problems in the adventure. In every single one of these encounters, a chat menu appears, from which possible lines can be chosen. These lines are the only ones that can be used in the communication with the other game characters and the only thing the player can do, is to choose which of the lines is to be used first. The player cannot make up own lines or ask the questions in its own formulated sentences, but only pick one line at a time from the already formulated chat menu.

In the booklet that comes with the game, the authors of the game initially make the following statement: (our own translation)

“The Longest Journey is an adventure game. You don’t fire a gun, you don’t blow up bombs, you don’t fight any monsters... wait, there are monsters. But unlike other games, your most dangerous weapon and your greatest asset in The Longest Journey, is your brain. All adventure games involve solving puzzles, exploring new worlds, meeting and talking to strange characters, both friendly and not quite so friendly, and finding out how to get out of the mess this time.”

Since the authors themselves claim that the player’s brain is the greatest asset, we think that it would be a better idea to actually let the player use this asset in the encounter with the different characters. This is also why it would be better if the player could make up her own questions, since this would make the player more involved in the game. This would in turn make the game more fun, interesting and also more difficult, since it requires more attention from the player, in order to remember what previous characters have said, when meeting new characters.

We therefore want to make an alternative dialogue approach, where we would allow the player to interact in her own language with a game character. We have picked a scene from the actual game and translated the dialogue, to have as a starting point for our own work. In order to make our chosen dialogue’s content clear, we first must explain the basics of the game itself.

The Story

The setting of the game is two parallel worlds, Stark and Arcadia, which are two very different places. Stark is the logical world; the world that the humans live in, while Arcadia is the magical world, where science has no place. Arcadia is full of different creatures, some human-like, some beasts. From time immemorial, there has been a balance between science and magic, order and chaos, Stark and Arcadia. For thousands of years, the Balance has been the guarantor of cosmos' equilibrium and the keeper of order between the twin worlds. But when the game is starting, a time of great anxiety has begun and Chaos is threatening to disturb the balance that should exist between the worlds.

The guardian of the balance has deserted his throne and a troop of very powerful evil men is trying to take the power. Only one person can restore the balance and save the worlds. This person has the incredible and rare ability to shift between the worlds. Her name is April Ryan and she is the one to take on the longest journey.

In the beginning of the game, the player (who takes on the role of April Ryan) does not know much about the dangers that are threatening the world in which she lives. She does not know that there is another world called Arcadia and she does not even know that the world she lives in is called Stark. However, she can feel something is going terribly wrong in her world, because she is having nightmares, where she sees awful sights and gets a feeling that what she is dreaming is actually a prophecy of what is going to happen. After a while, she also starts seeing things when she is awake. It is not just her imagination – weird things are actually happening in the world.

After talking to a number of mysterious people, April finds out that there is another world that exists in a parallel universe. The two worlds used to be one, but after many historical problems, there was a partition of the worlds. She also finds out that her own world is known as Stark. Stark's twin world, Arcadia, is one of great magic and what April has experienced during the last few weeks is parts of this magical world breaking through into her own, logical world. The cause of the weird images coming into Stark from Arcadia, is that the balance that used to keep the order between the worlds, has been disturbed by an attempt from a council of evil men to take over the power of the two worlds.

April must now shift between the two worlds, in order to restore the balance. There are many different tasks she has to perform before the balance can be restored. For example, she must find a secret plate, two magical dragons, the new guardian of the balance and the tower where the guardian will watch over the balance. She knows this much, when she makes her first shift into the magical world. When she arrives in Arcadia for the first time, she meets a character called Minstrum Tobias and he is one of the supporters of the balance. He is one of the chosen keepers of the balance, whose task it is to make sure that there always is a guardian watching over the balance.

The Scene and the Dialogue

The scene that we have chosen, is the encounter between April Ryan and Minstrum Tobias. We chose this scene mainly because it involves many interesting questions about persons and things, which April needs information about. The full original dialogue can be viewed in Appendix A. Note, however, that we translated the dialogue into English, since the game is in Swedish.

To make our idea work, we needed to change some of the dialogue lines and simplify the sentence structures. This was done to limit the amount of grammar rules and also to speed up the generation of answers that the game character will give the player. We used the original dialogue as a model for our game dialogue, but rewrote many of the character's lines. But even if the lines now look much more alike and shorter, we find that all of the important information in the original dialogue is kept intact. We also focused on the most important nouns, that April in this particular scene is interested in finding out information about. The result of our changes to the original dialogue can also be found in Appendix A. Here is one example of how we simplified the structures of some of the lines:

Minstrum Tobias:

“When the earth was divided and the land of the Guardian was created, a plate was made. The plate would serve two purposes, as key to the tower of the balance and as replacement for the plate that already existed in the tower, should that one be destroyed. The tower is now abandoned and locked and the old plate is broken. I think the time has come to find the new plate and use it.”

This was changed into:

The plate was divided in four parts.

The parts were given to four enchanted people.

The plate can restore the Balance.

The plate is the key.

The plate opens the tower.

3.2.2 *WordNet*

The use and understanding of specific and general terms has been seen in some of the research described in Chapter 2. Eliza's greatest asset was perhaps her ability to connect words with more general ones, making it look like she really understood what the user was talking about. For Eliza, these hyponym/hypernym pairs were hard-wired. In SUMMARIST another method was used. Instead of hard-wiring the connections between different words, SUMMARIST makes use of WordNet for its interpretations and rephrasals of topics.

WordNet is an electronic lexical database, used by researchers in the fields of computational linguistics, text analysis and other related areas. The design idea of the database is taken from current psycholinguistic theory and is based on the way human lexical memory is organised. Nouns, verbs, adjectives and adverbs are organised into synonym sets that represent one specific lexicalised concept.

The current version of WordNet contains about 92,000 synonym sets, which is quite remarkable since the whole database has been constructed manually, without any automatic acquisition methods used. Since one single word might have several different meanings and also belong to different parts of speech, most words in WordNet have more than one sense. There are even words that have over 10 different senses, thus belonging to different synonym sets.

Although synonymy is an important semantic relation between word forms, the semantic relation that is most important in organising different words, is a relation between lexicalised concepts. This is especially important for nouns, which make up for over 80 % of the entries in WordNet. It is the relation of subordination, which in this context is called hyponymy, which semantically links the different synonym sets together. For example, a *pine tree* is a hyponym (subordinate) of a *tree*, and at the same time *tree* is a hypernym (superordinate) of *pine tree*. It is this relation that organises nouns into a lexical hierarchy. There are also similar relationships between the synonym sets for other parts of speech.

This is an example of a lexical hierarchy in WordNet, constructed as a trail of hypernymically related synonym sets:

```
{pine, pine tree, true pine} > {conifer, coniferous tree} > {gymnospermous tree} >
{tree} > {woody plant} > {plant} > {life form, organism} > {entity}
```

The on-line version of WordNet 1.6 can be viewed and tested at (WWW4). However, in our work we will not use the on-line version. Instead, we will take advantage of the equivalent Prolog database. Here, WordNet is constructed as different Prolog files and predicates, which are all semantically linked in the same way as in the on-line version. Each Prolog database file contains information corresponding to the synonym sets and word senses contained in the WordNet database. In the Prolog version of the database, the `synset_id:s` (defined below) are used as unique synset identifiers.

Each line of a file contains an operator that corresponds to a WordNet relation. All lines with the same operator value are stored in the file `wn_operator.pl`.

The general format of a line in a Prolog database file is as follows:

```
operator(field1, ... ,fieldn).
```

Each line contains the name of the operator, followed by a left parenthesis, a comma-separated list of fields, a right parenthesis, and a period. There are no spaces, and each line is terminated with a newline character.

Each WordNet relation is represented in a separate file by operator name. Some operators are reflexive (i.e. the “reverse” relation is implicit). So, for example, if X is a hypernym of Y , Y is necessarily a hyponym of X . In the Prolog database, reflected pointers are usually implied for semantic relations.

A pair of `synset_id:s`, in which the first `synset_id` is generally the source of the relation, represents semantic relations and the second is the target. The main predicate `s/5` has the following form:

```
s(synset_id,w_num,'word',ss_type,sense_number,tag_state).
```

A `s` operator is present for every word sense in WordNet. In the main file `wn_s.pl`, `w_num` specifies the word number for word in the synset. Here is an example, taken from the file `wn_s.pl`. The two words are taken from the same synonym set, and therefore they have the same `synset_id`, but different word number.

```
s(100001740,1,'entity',n,1,1).  
s(100001740,2,'something',n,1,0).
```

We will use WordNet in two different ways when searching the KB for information on the topic given by the user. Since WordNet is based on synonym sets, we can simply use the `synset_id:s`, when searching for synonyms to the topic introduced by the user. If the user types in a word, that does not exist as a key in our KB, we can then check if there is a key in the KB that is a synonym to what the user is asking about. This will increase the number of keywords in our KB, without us having to add extra entries for each synonym. It will also help in situations where the player might not remember the exact term when questioning a specific topic, since any word meaning the same thing can be used. If the player is wondering about the position of the ‘entrance’, but types in the synonym ‘entry’, which is not in the KB, the system will still understand this word, since it has the same `synset_id` as one of the keywords in the KB.

The other way in which we will use WordNet, is by taking advantage of the hierarchical structure of the different synonym sets. We will allow the user to use subordinate words, compared to those in the KB, when asking the game character for information. This means that we will search for hypernyms to the topic, when the questioned topic itself is not present in the KB. Using the same example as above, this means that the player can use the word ‘door’, when asking about the position of the ‘entrance’, since ‘door’ is a more specific kind of ‘entrance’. However, the search for hypernyms is more complex and time consuming than the search for synonyms,

since each entry in WordNet has so many different senses and we cannot tell immediately which sense is the intended one in each situation.

In our example, the word 'door' has six different senses, each with its own chain of hypernyms. When searching for hypernyms that match a keyword in our KB, we have to go through each sense one at a time, going upwards in the hierarchy. The word 'entrance', which is a hypernym of the word, is present in the hypernym chain of second sense of 'door'. This means that we will have to go through the first sense's chain, before we are on the right track. In this case, it will not take that much time, but in cases where the correct word might be in the last of 10 senses, it will take up to as much as 30 seconds to perform the search.

In spite of this disadvantage, the use of WordNet will improve the overall performance of our system, since the system's flexibility will increase as we do not force the player to use specific terms when asking about different things in the game.

3.2.3 QTAG

QTAG is a language independent part of speech (POS) tagger written in Java by Oliver Mason at the University of Birmingham. It consults a tagged corpus and a lexicon, for any chosen language. The lexicon hands all possible tags and frequency for a word in isolation. The corpus is divided in trigrams and hands a statistical number for the input word to appear as a specific POS in a given context. The input words are run through a three word frame, where the individual words' all possible tag combinations are evaluated, referring to both lexical and contextual occurrence. The three highest scorings for the input word's POS are displayed as output from the tagger, together with the word itself.

The tagset used by QTAG is the Birmingham-Lancaster Tagset, BLT. A complete list of the tags can be viewed at (WWW1).

If an input word cannot be found in the lexicon, its three final letters are looked up in the lexicon to guess what POS the word could be. Mason (2000) gives an example:

The input word 'hiking' cannot be found in the lexicon. 'ing', its last three letters, are searched for as a suffix in the lexicon's entries. Let us say these words, with tags and frequencies, are found:

- king NN 23
- sing VB 15
- ring NN 2 VB 5
- walking VBG 10

The guess list for the input word ‘hiking’ would then be the following, which tells us that ‘hiking’ is most likely a noun:

- hiking NN 25 VB 20 VBG 10

In order to find the nominal heads and wh-words in the input, we use QTAG to tag the user’s input. Using QTAG, we can list the nouns of the input to define topic, and the wh-words telling what kind of information is requested. Small changes in the program code will be necessary, and also an implementation of the interchange between the Prolog and Java code. A condition for making this interchange work, is the use of SICStus Prolog’s library for calling Java from Prolog. Conveniently, SICStus Prolog offers a library called Jasper, which we have used for this purpose.

QTAG is freely available for research purposes from (WWW2) and with it follows a tagged corpus and lexicon for English. These are the resources we use, running QTAG in our input analysis.

3.3 Input Analysis Method

Since we want the player of the game to have the freedom to formulate her questions to the game character in any way she wants, we have chosen to disregard any syntactic analysis of the player’s input to the game character. We have also chosen not to use any methods for semantic analysis of the input. We decided at an early stage in the project to devote more time and energy to the output generation than to the input analysis. For this reason, we need a fairly simple method for analysing the player’s input to the system, although not as simple as matching the user’s input to certain patterns, as in for example Eliza and Julia.

As mentioned earlier, previous research on the subject of language analysis, has shown that it is very difficult to say what the meaning of a single, isolated utterance is. With this knowledge in consideration, added to the fact that we want to keep the input analysis as simple as possible, we have decided to go along with Clark’s idea (1992) that what is most important in a conversational contribution generally is a noun phrase that the speaker introduces. We have therefore chosen to search for important nouns in the player’s input question, to see if the game character knows anything important about these nouns.

The input analysis will have to perform the following tasks:

- tag the input words with their POS,
- find and list the nouns,
- find possible wh-words.

The purpose of the input analysis is to define a list of possible topics in the player’s input, and what sort information she wants, which is done by checking the wh-words.

The definition of the current topic is not done in the input analysis, but in the inference machinery, connected to the KB, comparing the list of possible topics with the keys of the KB.

3.3.1 Definition of Topic

Since definition of topic within one single sentence is such a complicated task, we need to find a good and simple method for topic identification. The best ad hoc solution to be found, seems to be Allwood's and Andersson's (1996) observation that the topic most often coincides with the subject of the sentence, which is almost always a content word. Clark (1992) seems to agree with Allwood and Andersson, since he claims that a speaker introduces the topic by the uttering of a noun phrase in its first contribution. In our situation, the noun phrase introduced by the player will be accepted by our system and defined as topic if it matches a keyword in the KB. Using Clark's terms, the existence of the topic in the actual game will be common ground between the player and the system.

We have therefore decided to follow Allwood, Andersson and Clark and focus on content words and narrowed it down to only words that can be the head of nominal phrases, i.e. proper nouns, pronouns and nouns, which we from now on will refer to as 'nominal heads'. This decision is based on the fact that **things** play a central role in the game. It is the things that lead the player to the solution of the game and therefore most questions are likely to be about these things. We are aware of this limitation of the input analysis method, but we still find it necessary to do.

3.3.2 Distinguishing Wh-questions

There are very roughly three kinds of questions in English: yes/no-questions, questions containing wh-words (where, what, why and more), and statement questions where the syntax follows an imperative phrase, such as "Tell me about the tower".

Our system only makes the distinction between wh-questions and questions without wh-words. This is mainly for the need of making at least one distinction. If for example the user asks "Where is the tower?" it would be unsuitable to reply with "The plate opens the tower.", which is some of the information on the keyword 'tower' in the KB. It is relatively easy in the analysis to search for these wh-words, compared to the amount of work that would have been required to define a yes/no-question. If the user asks "Do you know where the plate is?", the system will not analyse this question as demanding a 'yes' or 'no' as an answer, but as a request for information on a noun, i.e. is the plate. Implicit questions are solved in this way and we hereby satisfy Allen and Perrault (1980) saying that a yes/no-question is a request for not just 'yes' or 'no', but also some implicit information.

3.4 Knowledge Representation Method

3.4.1 The Choice of Knowledge Representation

There is a close relation between how one chooses to store the knowledge and the design of the generation module. Most systems use some kind of semantic representation in the KB and before sending this representation to the generation module, some other processes are performed, where the propositions from the KB are given a deep structure. The deep structures are then input to the generation system. These systems are called automatic generation systems (Danlos 1987). Systems which in the KB represent knowledge containing information on what words and in what order they should be in the produced sentence, are according to Danlos, not called automatic.

There are, however, many ways to represent knowledge. In previous work, we have used FOL, first order logic, to represent information and to translate all input given to the system into FOL, mapping this against the KB. FOL has turned out to be sufficient for a small domain and only for a limited variation of lexical input.

Studying different ways to represent knowledge, it all comes down to Danlos' (1987) observation: "The relationship between language and knowledge representation is by no means simple. In practice, specialists tend to use a great variety of representations. They usually select a particular representation because it suits their particular goals." With this in mind, we have chosen to represent the knowledge in our KB by using English words. We motivate this choice, not only by the explanation that it suits our particular goal, but by using English words in the KB, we need no process where semantic representation becomes deep structure. We save some work and simplify the process. Taking in consideration Danlos' warning of representing knowledge in natural language (see Section 2.4), we have no intention of making a multilingual text generation system. When it comes to variety, we claim that this can be achieved in other ways, for example by using WordNet in the generation process.

Our knowledge representation module (KR) is divided in two parts: the KB containing the game character's knowledge and the inference machinery containing functions for processing and retrieving knowledge. Figure 2 shows an overview of the Knowledge Representation.

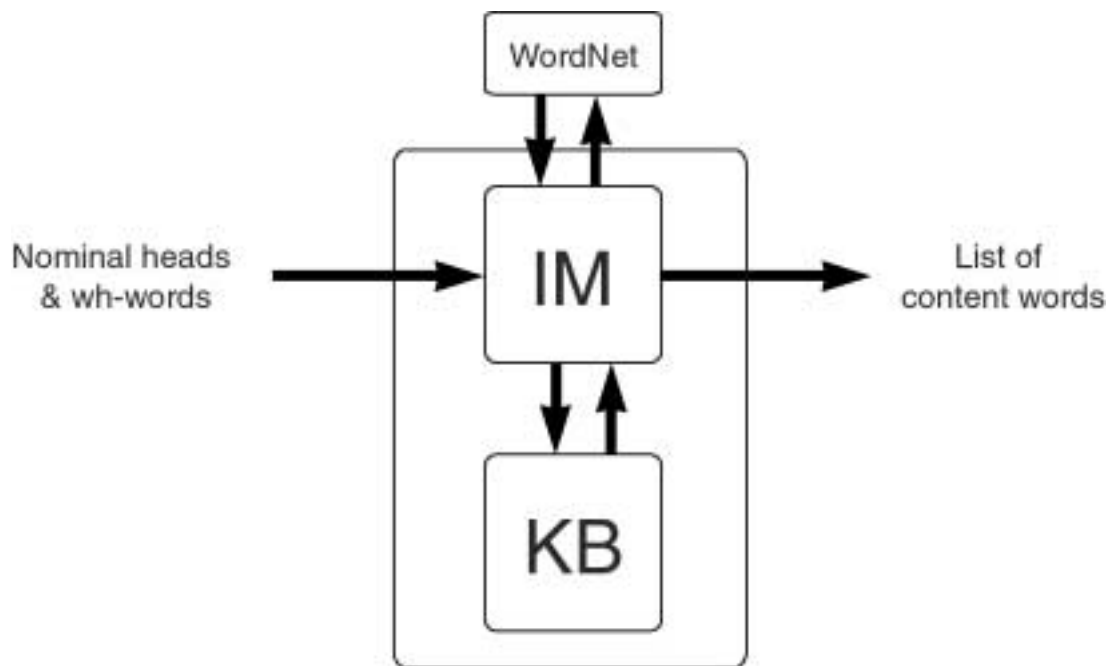


Figure 2: An overview of the Knowledge Representation module, showing how the Inference Machinery communicates with the Knowledge Base and WordNet.

Our inference machinery will:

- Select possible wh-word from the input.
- Compare the list of input nominal heads with the keywords in the KB. If no match between keywords and input, check match for synonyms, hypernyms and anaphors. If a keyword is matched: define this as topic. If no keyword is found, topic is set to be anonymous.
- Select the information stored in the KB for the defined topic and wh-tag.
- Send the selected information (list of content words) to the generation module.

3.4.2 The Knowledge Base and the Inference Machinery

We have decided to implement the KB as a data structure that is similar to a hash table. The topics on which the system holds information, such as 'plate', 'tower' and 'guardian', are the keys in the hash. The values are the list of words that the output sentence should be generated from.

The task of the inference machinery is to define the topic of the input from the player. We have chosen not to perform this task in the input analysis, which perhaps is the most common, since our definition of topic is a word that (directly or indirectly) occurs both in the input and as a key in the KB. If we had chosen to define the topic in the input analysis, the input analysis would have required access to the

KB and since we want to separate the modules as much as possible, we cannot allow this.

3.5 Sentence Generation Method

For the output generation of sentences, we decided to write a simple DCG (Definite Clause Grammar) and a lexicon containing all of the words in the chosen dialogue. Using only the grammar and lexicon, we found that the system generates perfect sentences as far as syntax goes, but semantically the sentences are ill-formed since all possible sentences can be generated using only the grammar as generation method. Taking the content words, putting them through the generator, the character then has a reply to the given input. Since one single input sequence can correlate to many different outputs, the character has to be able to choose one of them as its response. This can be seen as a human feature – not to respond in the exact same way in different contexts for one specific input, and this gives the impression of a wider intelligence than just mapping one specific answer to another specific input, like Julia or Eliza. Inspired by the Compansion Project (1992), we wanted to use a method that generates only semantically correct sentences from lists of content words. In this section, we will describe this method.

3.5.1 An Overview of the Sentence Generation

As mentioned, the idea for our sentence generation module comes from the Compansion Project (CP), 1992. We will generate full English sentences from lists that mainly are made up of content words. CP uses a complex lexicon where every entry is tagged manually with both semantic and syntactic features. This lexicon is CP's greatest advantage as it prevents the grammar to generate ill-formed sentences. However, the huge lexicon is also a big drawback, since expansion of the domain and lexicon would be very difficult and time consuming. This is also why we have chosen not to make a complex lexicon, but to keep the lexicon simple with few features and instead focus on the generation method for preventing semantically ill-formed sentences.

The CP system users do not have to type the words in the intended order, since the lexicon includes syntactic information on what positions and functions the different words can have, putting only the correct sentences together. Since we will not include that many features for the words in our lexicon, the input words to the sentence generation module will have to appear in the intended order. Given that the input to the sentence generation module in our system always comes from the KR module, we assume that the composer of the KB will be competent enough to put the

words in the correct order⁸. For us, the possibility to easily expand or replace the lexicon in order to implement more than one character of the game, is more important than a complex lexicon. The need of a post-processing module, filtering unwanted output, will however be necessary.

The generation module will take a list of content words as input and will then generate a full English sentence from the words. Since the input is sent from the KR module, we will be able to control the semantics of the generated sentence. The syntax is controlled by the grammar, which is used in the generation algorithms. If the player asks a question like “Where is the plate?”, the previous modules will have analysed the question and a search for a correct answer will have been performed. In this case, where our system has information that matches an answer to the question, the generation module will receive this information as a list of content words ([plate,arcadia]). The generation module will then make a sentence from these words and return an answer: “The plate is in Arcadia”.

When the input to the text generation module is an empty list, i.e. when the KR’s inference machinery has found no knowledge that matches a possible answer to the player’s question, a separate predicate will be called without using the generation module at all. The predicate uses a list of hard-wired output sentences, randomly choosing one of them. Some of these sentences, put to the user are “Sorry, I cannot help you with that”, “Tricky question, ask someone else” and “Anything else you want to know?”.

3.6 Post-processing Method

In order to polish the system’s output, and sometimes also to filter the generated sentences so that only the most appropriate ones are shown to the user, we will implement a post-process. The post-processing module will contain different functions that in some way will improve the output from the generation module. The different functions in the post-processing module will be correction functions, textual awareness functions and print functions.

The correction functions will discard sentences that for some reason not should be viewed by the user. For example, our lexicon does not include the article ‘an’, only ‘a’, and in the post-process, we have to change ‘a’ to ‘an’ where necessary.

The textual awareness functions will take generated sentences as input. It will then put these sentences together, making them look more like a text. Inserting anaphors and conjoining sentences does this.

⁸ This differs from the CP, but that system was designed to be a handicap aid, which our system is not.

The print functions will be necessary as we are handling sentences as Prolog lists. For example, we do not want an output from the system look like [i,am,the,librarian], but rather as “I am the librarian.”, where all sentences start with a capital letter and end with punctuation and where the personal pronoun ‘I’ always is capital, even within a sentence.

4 Implementation

In this chapter we will describe the full implementation of our system. We have chosen not to implement any graphical interface, mainly because our system is not meant to be an alone-standing program. It is intended to be used within the adventure game and possibly used with a speech interface, so that the player can talk to the implemented character. The game itself has to provide the speech recognition and synthesis as we only treat written input and give written output. Our system's input does not have to be a grammatically correct sentence, only the single words are needed, which a speech recogniser should be able to distinguish. Our system's output has to be on a form that easily could be put to a speech synthesis program, e.g. English sentences.

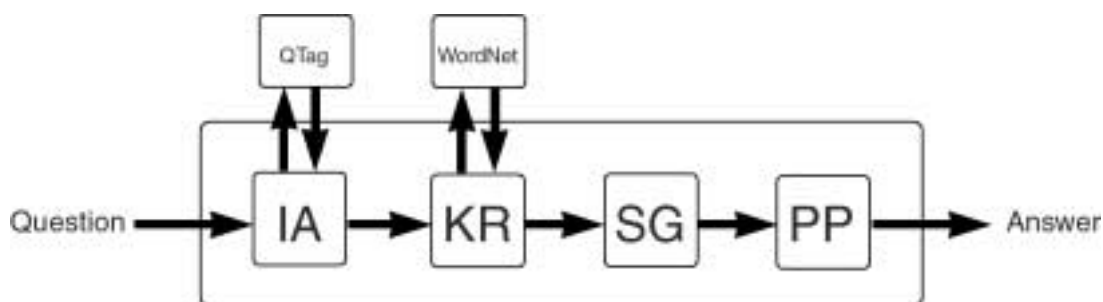


Figure 3: A system overview showing input and output and the connections between the modules (Input Analysis, Knowledge Representation, Sentence Generation and Post-Processing) and the resources QTAG and WordNet.

We will in detail discuss the four modules shown in Figure 3: input analysis, knowledge representation, sentence generation and post-processing.

4.1 The Input Analysis Module

The input analysis' task – to tag the user's input and distinguish wh-words and nominal heads – can actually be divided into some subroutines. We will here outline these subroutines, which are also shown in Figure 4, below.

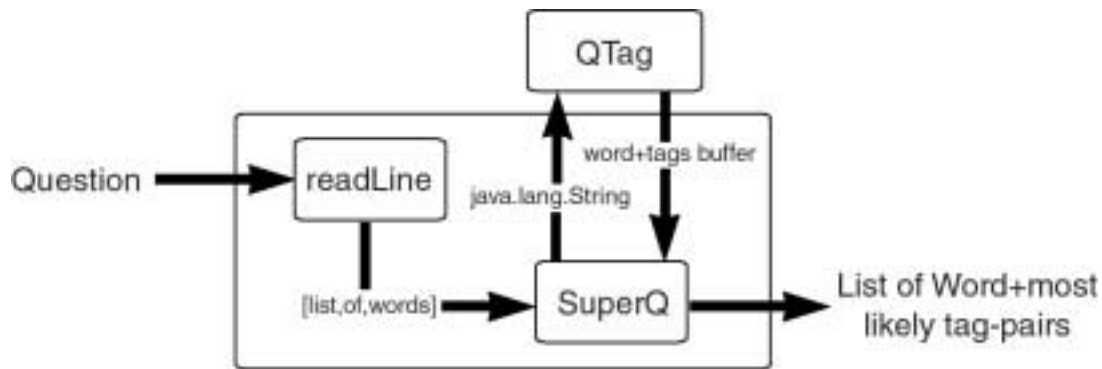


Figure 4: An overview of the Input Analysis Module.

4.1.1 Reading Input

Reading the user's input from the screen, we use a predicate called `readLine/1` (Blackburn and Bos, 1998). The predicate reads characters from the screen, clustering them into words that make up elements in the output list. Punctuation is discarded and capital letters are made small, in order for Prolog not to interpret the words as variables. If the user writes "I am looking for a plate", `readLine/1` will represent this as `[i,am,looking,for,a,plate]`.

4.1.2 Calling Java and QTAG

Using QTAG, that actually takes a string as input and returns a buffer, we decided to write a new Java class, `SuperQ`, in order to manipulate both input and output. The class `SuperQ` therefore takes a list of words from `readLine/1` as input, transforms the list into a Java string and calls QTAG with this string. The output from QTAG is a Java buffer, containing the input words, their possible POS tags and the probability for each of these tags. `SuperQ` now transforms this buffer into a Java string with only the input words and the most probable POS tag for every word. The string is on the form "word POS word POS ...". which is also the output from `SuperQ`.

4.1.3 Polishing QTAG's Output

In Prolog, we now need to transform this string back into a list, where a word and its POS are paired to a list element. Doing this, the list will look like `[(word1,pos1),(word2,pos),...]`. However, some further changes are also necessary.

Testing QTAG with different input, we found that the word 'where', which is an important word for the user of our system, always gets the tag of a subordinated conjunction, CS. This happens even when the word occurs first in a sentence, as for example in "Where is the plate?". Consulting Oliver Mason, the creator of QTAG,

the reason seems to be that in the consulted corpora, the frequency of where the word actually appears as a subordinated conjunction is so high that it overrides the contextual influence. Mason (2000) himself claims that there is not much one can do about this, but possibly to implement your own post-process, taking care of these situations. To follow his advice, we converted all CS tags given by QTAG to our tag 'wh'. This means that a word such as 'until', which by QTAG gets the CS tag, will in our system be retagged as a wh-word. This is obviously not a brilliant solution, though appropriate for its task. In the KB search, our system ignores words that are tagged with 'wh' if they are not 'where' or 'who', since this is the distinction most relevant in our chosen game situation. More about this later on in this chapter. We also had to hard wire the tag of 'Stark', the name of one of the worlds in the game, giving it the 'pn' tag instead of, as QTAG does, tagging it as an adjective. This means that the English adjective 'stark' will never be correctly analysed by our input analysis. Again, this is not a perfect solution but, with access to a larger corpus, QTAG would not have made this mistake in the first place.

The different tags in QTAG consist of capital letters possibly followed by a \$-sign. Since we needed to connect the output from QTAG to the rest of our system, implemented in Prolog, we had to make some modifications here. The \$-sign will not work in a Prolog program, since Prolog identifies all names starting with a capital letter as variables and recognises odd characters as syntax errors. Our intention was therefore to modify QTAG's output, so that all relevant tags are changed into our own tagset. We have chosen to do this outside QTAG, wishing to keep the original program code of QTAG intact.

Since QTAG uses BLT, these tags are not the same as the tags we use in our program, namely those we use defining the words in our lexicon. For example, we only use one tag for different kinds of pronouns, where as QTAG names them differently. We therefore convert the QTAG tags to our more general ones by listing the interesting QTAG tags and our own translation for them. Note that only tags that are interesting for our system are converted, all other QTAG tags are kept.

In TEXT (McKeown 1985), a compound noun is represented with an underscore between the words and saved as one unit. This could be done since TEXT does not really contain any input analysis, and the fact that an arbitrary user would not naturally type compound words this way, did not have to be considered. In our system, we have no need to handle compound nouns since none of the keys of the KB are compound. It could be, that the player uses a compound noun that actually is a synonym or hyponym to one of the keys. Using QTAG analysing the input, we solely rely on its capability, not implementing any compound word recognition of our own.

The output list of pairs, with the wanted tags, looking like [(word,POS),(word,POS)], is sent to the KR. This means that all input words are kept and sent to the KR, not only those interesting for our system (such as nominal heads and wh-words). This

distinction is therefore not made in the input analysis, but by the inference machinery of the KR.

4.1.4 Java and Prolog: a Problematic Friendship

The interchange between Java and Prolog, which needed to be solved, meant calling Java from Prolog, since our main program is written in Prolog. This interchange was by no means as simple as we first expected. SICStus Prolog provides the library Jasper, which is a library of modules for this purpose. There are two ways of calling Java using Jasper, and we started with the one that appeared to be the simplest one. After weeks of trial and error, though strictly following SICS' personal guidance, a bug fix version, for the operating system we use (Windows 98) was sent to us. Testing this, without success, SICS informed us that additional software has to be installed in order for this way of calling Java to work. This software, we did not have any license for.

We therefore tried the second way of calling Java from Prolog, and after several weeks of trial and error, we realised that the Java compiler we were using, was actually too new for combining Java and Prolog. An older version of JDK was therefore installed. Unfortunately, more days of frustration were still to come. In late May, after months of e-mail correspondence with SICS' personnel, the final installation is made: a newer version of SICStus Prolog. At last, we could call a Java program from our Prolog system. This also meant that QTAG at last became an integrated part of our whole system.

In summarisation, what we needed for this Java call to work, was patience, SICStus Prolog 3.8.2, JDK 1.2.2 and SICS' helping hand – nothing else is to be recommended if you want to avoid these four months of frustration.

4.2 The Knowledge Representation Module

The knowledge representation module (KR) of our system does not only contain knowledge stored in the knowledge base (KB), but also methods for retrieving and using the knowledge, the inference machinery. We will here in detail show how our implementation of the KR module works, which also can be seen in Figure 5. In Section 4.2.5 we will guide you through the KR following an input example.

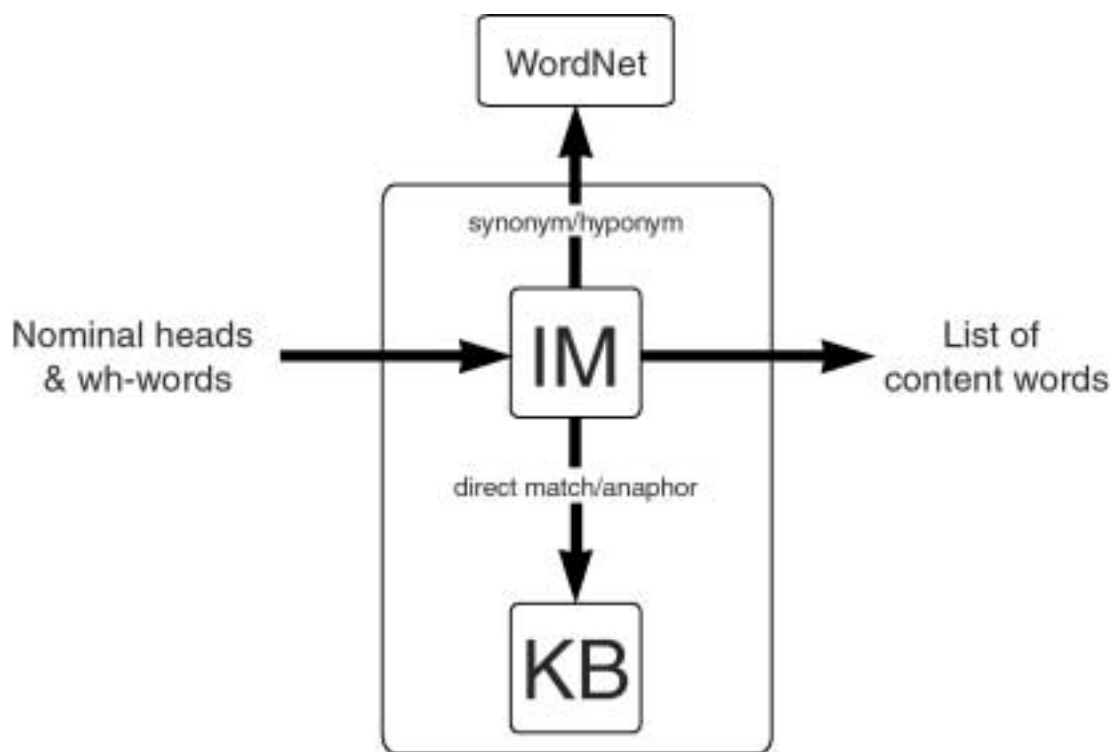


Figure 5: An overview of the Knowledge Representation Module, showing the four different methods for topic identification.

4.2.1 Storing Information in the KB

To know how to design the KB, we have to know what information it needs to contain. We early decided to skip a semantic representation of what is true and what is false, and instead to build this into the KB. We have chosen to store the information in a structure quite similar to a hash table. The topics of the game, on which it is assumed the player wants to receive information, are stored as the keywords of the KB. It is only about these topics that the system can give information. The information is stored as compound lists of which it is possible to generate English sentences. There is a distinction between general and specific information. The general information is given as an answer to a non-specific question, such as “Tell me something about the tower”. The distinction between this kind of general questions and more specific questions about the position of things, as in the example “Where is the tower”, is necessary to make. Without this distinction, defining all information as general, an answer to the latter question could be “The guardian must return to the tower”. The KB’s information is therefore tagged with what kind of question it answers.

The specific tags are ‘where’ and ‘who’ and all other kinds of information are tagged as general information – ‘g’, which can be the answers to general questions about a certain topic. There are many reasons for distinguishing only between these two wh-

questions. As mentioned earlier, the positioning of things is an important issue in the game, and we therefore found the ‘where’-question to be very interesting and necessary for the player of the game. The ‘who’-question is also important, since the player needs to find out information about certain other characters in the chosen game situation. But we did not choose these two wh-questions simply because they are interesting and important. They are also quite easy to implement, since the use of those words in a particular question quite definitely determine what type of answer is to be expected, following a question containing either of the words. A question containing ‘where’ normally asks about the position of a thing or person and a ‘who’-question usually is followed by a description about the person in question. We considered distinguishing between all kinds of wh-words, but soon realised the difficulty of the wh-word ‘what’. There is no easy way of telling what is to be expected in an answer to a question containing a ‘what’. Since we do not analyse the full semantics of the input questions, but only keep the wh-words and the topic noun, we cannot easily tell what kind of information a player that has just asked ‘what, plate?’ wants to have. We could guess that the player wants to know what the plate is, but the words could also have come from a question such as “What can you tell me about the plate?”, “What is the plate used for?” or “What building is the plate stored in?”. For this reason, we have chosen to ignore the other, more difficult, wh-words and only distinguish between ‘where’, ‘who’ and ‘g’ (general) questions.

Since the information on the various topics are of varying kinds, it is not always the case that all topics have an answer to the question ‘where’ or ‘who’. What specific information there is, depends on the topic and what information the character holds on this topic in the real game. This shows a sample from the KB, showing what information is stored on the topic ‘guardian’.

```
kb(guardian, [(where, [x,disappeared,in,stark]),
              (who, [x,protects, balance]),
              (g, [x,return,to,tower]),
              (g, [you,find,x])
              ]).
```

The sample shows that among the information on the topic ‘guardian’, there are two pieces of information that can be the answer to specific questions, namely those asking ‘where’ or ‘who’ the guardian is. The other information is tagged with a ‘g’ for general. The letter ‘x’ is a placeholder that tells us where in the sentence to put the topic in the generation of the answer. We found this insertion necessary to know what is the subject respectively object in information such as “You must find the guardian”. In the Compansion Project, (Demasco and McCoy, 1992) the selection of subject and beneficiary was left to the user, choosing between for example “Jack gives Mary the apple” and “Mary gives John the apple”. Our system cannot leave

this decision to the user, since the generated sentence has to be a true proposition about the actual game.

Letting the KB also represent the semantic world, the information also tells us what is true in the specific game situation. From the information on the guardian's position, namely that he has disappeared somewhere in Stark, we can draw the conclusion that all other information on the guardian's placement is false. The answer "The guardian is in Arcadia" will never be generated simply because among the information on the guardian's placement, Arcadia is never mentioned, and therefore it is false⁹. This goes along with what Allen (1995) and Reiter (1978) call the closed world assumption, CWA, namely that the KB only holds information about what is true. If a proposition about something cannot be proven from the KB, its negation is assumed to be true.

We mainly wanted the KB to consist of content words, or more precise nouns, verbs, and adjectives, in the same manner as the Companions Project (Demasco and McCoy, 1992) by generating full English sentences out of only these types of words. However, after the implementation of the generation module, which was designed to always generate as short sentences as possible from the given words, in order to prevent looping, we realised that we needed to include some prepositions in our KB. This was done in order to prevent a certain kind of incorrect sentences to be generated at a later stage.

The main reason for this is that our grammar includes both the rules $vp \rightarrow vbar\ np$ and $vp \rightarrow vbar\ pp$ and if the information in the KB does not contain any prepositions, the word list [balance, exist, worlds] can be turned into both "The balance must exist between the worlds" and as *"The balance must exist the worlds" in the generation module. In this case, where conflicting grammar rules can cause incorrect generations, we chose to put the preposition in the KB, which will lead to the latter sentence never being generated. Therefore, for some information, there are not only content words in the KB, but also a preposition. Another way of solving this would have been to add more features to the words in the lexicon, to distinguish between what types of complements different verbs take. We could also have chosen to make the grammar rules a little less general, but we felt since the grammar and lexicon worked perfectly, we could get around this problem simply by inserting very few prepositions in some of the entries in the knowledge base.

The full implementation of the information in the KB can be seen in Appendix B.

⁹ Our system cannot generate negation, meaning that from [guardian,disappeared,Stark], the sentence "The guardian did not disappear in Stark" will never appear.

4.2.2 Recognising Wh-words

The result of the input analysis is a list of words where each word is paired with its POS tag. The first process of the inference machinery in the KR module is to find a possible wh-word from this list. This is done in a predicate called `define_quest/2`. The predicate simply searches for words tagged 'wh', and only for the words 'where' or 'who', as explained earlier. This means that those subordinated conjunctions, mistakenly tagged 'wh', because of the problem with QTAG, will not actually be treated as wh-words. If there is no word tagged 'wh' in the input, the output of `define_quest/2` will be 'any', meaning that the user asks for general information on the topic.

4.2.3 Defining Topic

The next step is to define which of the possible nominal heads from the input sentence that is the topic of the current question. We treat this matter in a very user-tolerant way, since we have no syntax analysis of the input and therefore allow the input to be formulated very freely. This gives the user freedom to express the question in any way she wishes. If the user happens to mention a nominal head that appears as a keyword in the KB (even if she actually asked about something else) the system will generate information on this word. Just like in the Persona Project (Ball et al. 1997), we make no claim of creating an understanding system, even if our game character, like Peedy the Parrot, might give the illusion of understanding what the user is actually saying.

We start by trying to match nominal heads from the input with the keys in the KB. If there is no nominal head in the input that directly matches the keywords in the KB, the system starts looking up synonyms to the input nominal heads using WordNet. This is one of the systems flexibilities. We could hard-wire some synonyms to the KB keywords, but we choose not to. We want to use the dynamic and semantic relations that WordNet is based on. If no match between input synonyms and keywords can be made, we have also implemented the search for anaphora and hypernyms. A match between input and KB keyword is hopefully found, and the topic is defined as this keyword. If no match between the input nominal heads and the KB keys can be made, the topic is set to be anonymous. By this, our definition of topic is not necessarily what the input question is about, but what nominal head questioned in the input that corresponds to a keyword in the KB. If the user's input happens to contain more than one nominal head that could be defined as topic, only the first mentioned will be treated by our system. More on this in a later section.

There are four possibilities for the input word to relate to a KB key: direct match, being a synonym to a key, being an anaphor to the previous defined topic, or being a hyponym to a key. The search is performed in this order and if the input contains more than one nominal head, all of them are checked before trying the next possibility. This means that for the input "Tell me about the plate", with the two

nominal heads ‘me’ and ‘plate’, ‘plate’ will be defined as topic since this directly matches with the keyword ‘plate’ in the KB, even if ‘me’ could be an anaphor for the previous topic.

Using these four different search methods, it is possible for the user to have the following conversation with the system. The underlined words show examples of the different possible references. (Note that the KR module does not take sentences as input and does not give full sentences as output. This is only to show the KR’s part of the whole system.)

USER: Who is the keeper? (hyponym)
SYS: The guardian protects the Balance.
USER: Where is he? (anaphor)
SYS: The guardian disappeared in Stark.
USER: What is Stark? (direct match)
SYS: Stark is the logical world.
USER: Where is the entry? (synonym)
SYS: The entrance is in the tower.

The four different ways of identifying the topic of the input are described in detail below.

Direct Match

When the input contains a word that itself is a key in the KB, this word can immediately be defined as the topic of the input sentence. We have limited the search only to apply on the keywords of the KB. It is not possible to match the input words against the information inside the KB. We have not implemented any morphological analysis, which means that conjugated forms not in the KB will not be matched. For input in English, as here is the case, a simple but definitely not complete solution, could be to understand nouns in plural or verbs in present tense third person, when they end with *-s*. We have chosen not to implement these functions, which could be seen as a drawback, but we find that it would require work in an area not in purpose for our thesis.

Synonyms

In WordNet’s Prolog database all words are given a semantic ID. A word that has many different meanings is given many possible IDs, but all words with the same meaning, i.e. synonyms, have the same ID, as shown in Section 3.2.2. We use this possibility to enable the user to be understood by using synonyms to the keywords in the KB. Our predicate for identifying synonyms, `findSynonym/2`, looks up the input word’s semantic ID in WordNet, list all other entries of the same POS that has the same ID. Then it checks if one of these words is a KB keyword.

Anaphors

The most complicated search method for references, is understanding of the anaphors. There has to be a match between the features of the pronoun being used as the anaphor and the features of the word it refers to. This requires a dynamic storage of the topic of every input sentence. After each input question from the user to the system, the defined topic has to be stored in a dynamic predicate, in case an anaphor occurs in the next question. If that happens, a comparison between the last topic's features and the anaphor's features has to be made. The predicate for identifying anaphors is findAnaphors/2.

Knowing a word's features requires access to a lexicon. We made the decision to make a separate lexicon for the KB, instead of using the same one as for the generation module. This is simply because we want to prevent changes in the lexicons to conflict with the different implementations of the two modules. A change in the lexicon for the sake of the KB search might then lead to a failure in the generation module and to keep this from happening, we instead use two similar lexicons in the different modules.

Another reason for having separate lexicons is that in the understanding of anaphors, we only need a lexicon with nouns, personal pronouns and proper nouns. There are also some features, for example gender, which are necessary in the understanding of anaphors but not in our generation process. Since the features of the words needed for generation and understanding anaphors differ, and we want to keep the different modules separated as much as possible, an extra KB lexicon is needed.

Two types of anaphors are understood by the system, namely when a personal pronoun refers to an earlier topic being either a proper noun or a noun. 'He' can therefore both refer to the topics 'Yerin' and 'guardian', which both have the features third person singular and masculine in the KB lexicon. The guardian cannot be referred to as 'it' since he has the feature masculine. Since only the latest defined topic is saved, it is not possible to refer to a topic discussed much earlier. If one question is about the guardian, and the next one about the plate, a third question about 'him' will not be understood as referring to the guardian. In fact, the reference will not be understood at all, since 'him' cannot refer to the current topic 'plate'. This goes against the fact, stated by Levinson (1993) that, in a human-human dialogue, a new topic can be introduced, without closing the previous one. The participants can then later go back to discussing the previous topic, referring to it by anaphors and still be understood. A stack structure could be used to implement this, where all topics are saved and anaphors are searched for referents by popping one at the time.

Hyponyms

The user can also question a word that is a hyponym to a keyword in the KB, which means that the user can use a more specific word in its question and receive information about a topic that is more general than the questioned word. The system will then, using WordNet's hierarchical semantic network, search for the input word's hypernyms and check if one of these is a keyword in the KB. We have chosen not to implement the other way around: it is not possible for the user to be understood by using a more general word than those used in the KB. We found that such a solution would make it too easy for the user, since asking about a human being, could give information on many various topics, for example the guardian or other persons. Since most words are linked in one way or another in WordNet, allowing the user to ask for a more general term than the keyword would lead to the user receiving information on all topics just by questioning the word 'entity', since this word is hypernym to almost every other noun. But if the user happens to know that she is looking for a disc, when the more general word 'plate' is stored in the KB, she should not be punished for using a more specific term. These are the reasons for understanding hyponyms but not hypernyms. The predicate is called `findHypernym/2`.

As we have implemented the search predicates, the program first tries to match input words with direct match. If this fails, it also looks for synonyms, anaphors and hypernyms, in this mentioned order. The search for hypernyms is unfortunately, as it is implemented now, very slow. This is because of the great extent of WordNet's Prolog database, where each word has a great number of different senses with different POSs and all of them have a large number of hypernyms. All possibilities of hyperlinks have to be checked before the predicate can end with failure and running the program, it can take up to 40 seconds to get an output. This happens when the KR module input does not contain a direct match, a synonym or anaphor that can be defined as topic. Therefore it also takes this long when the player misspells a word so that no topic can be found, or when the user asks something that the system has no answer for. We therefore add an extra output from the system, which is given to the user while the search for a hypernym is going on. This means that when the system starts a hypernym search, it types "Hm, wait a minute" to the user, indicating some time of thinking and coming up with an answer. This corresponds to the way in which people in a human-to-human conversation keep their turn in the conversation, since the game character here does not let go of control over the conversation by abdication or repetition (Walker and Whittaker, 1990).

4.2.4 Selecting Information

Depending on the input, we now have a topic and an optional wh-word. The next step of the process is to select information from the KB, given the keyword and wh-word. Unifying the defined topic and the KB keyword, and unifying the current wh-word with a information type tag does this. All of the stored information under this

keyword and tag is selected and will be put to the sentence generation module. As the producers of TEXT (McKeown 1985), we here assume that the user has no previous knowledge on this topic and therefore give all information available in the KB. If the topic is set to ‘any’, an empty list will be put to the generation module. The selection of information is done in the predicate searchKB/3.

4.2.5 A Guided Tour through a KB Search

In this section, we will guide you through the inference machinery and the KB, in order to explain how a search in the KB works. We will first show how the inference machinery treats the player’s input question “Where is the guardian?”, which is a very likely question from the player of the game. When this input reaches the KR module, i.e. after it has been tagged by QTAG with the words’ POS, it will look like (1). The list contains of pairs of the words and their disambiguated POS. This is the input to the inference machinery and the KB.

(1) [(where,wh),(is,v),(the,det),(guardian,n)]

The first thing to do is to find a possible wh-word in the input. This is done, as explained earlier, in order to know if the player wants specific or general information on the topic. Since we in the KB only have specific information for the question words ‘where’ and ‘who’, these are the only words that should be wh-tagged, but QTAG naturally tags all English wh-words with the ‘wh’ tag. Therefore, we exclusively search for the words ‘where’ and ‘who’ tagged with ‘wh’ in the input. No other words with a wh-tag are considered, meaning that a question like “What does the guardian do?” will be treated as a general question. Because of the order of the Prolog predicates in our inference machinery, we first search for ‘where’, then ‘who’. This means that if both of them occur in the input, ‘where’ will be the chosen one. If neither of them occurs, the question word will be set to ‘any’, indicating that the input question is a general one.

We now have to define the topic of the input question. This is done in a predicate called define_topic/2. As explained earlier, we have made great restrictions from the usual meaning of topic by saying that we consider a topic being a word that most often is a noun but also a word that can be the head of a nominal phrase, i.e. a pronoun or proper noun. As described earlier, it is possible to do different searches in the KB looking for either explicit words (direct match), synonyms, anaphors or hypernyms. This means that the topic of the input to the KB can be one of these. The predicate define_topic/2 first excludes all words from the input that cannot be head of a nominal phrase, leaving only the possible words, excluding their POS, as shown in (2). It then starts the search in the KB.

(2) [guardian]

A predicate named `findKB/2` is called in order to find the actual topic. The list of all candidates, (2), is first searched for a word that explicitly is a key in the KB. Here, we find a match between the first (and in this case the only) candidate and a key in the KB also named 'guardian'. The topic is found, defined and saved in a dynamic predicate called `current_topic/1` in case of future anaphoric references.

Let us say we had received another input question from the player where the list of topic candidates was longer, for example (3), resembling a possible question from the player being "I wonder if you can tell me something about a dog or a guardian". Even in a case like this, 'guardian' would be defined as the topic, since all candidates in the list are checked for a direct match in the KB, before trying the other search methods.

(3) [i, you, me, something, dog, guardian]

This means that even if the previously defined topic from the preceding question from the player would correspond to one of the pronouns in the candidate list as an anaphor, a word matching directly to one of the keys in the KB is chosen as topic. If the candidate list contains more than one word that directly matches the KB, the first mentioned is defined as the topic.

We have now defined the topic, 'guardian', and the question word 'where' and we are ready to search the KB for information that corresponds to a question about the position of the guardian. We need to find a possible answer to this question, by checking if such information exists in the KB. This is done in a predicate named `searchKB/3`, where the input to the predicate is the topic and the wh-word. The output will be a list of words, taken from the value corresponding to the keyword 'guardian' and the wh-word 'where' in the KB. This list will later be sent to the sentence generation module. In fact, the output will be a list containing all of the information lists that can be given for this specific combination of topic and wh-word. The output for our earlier example, where the player asks "Where is the guardian?" will look like (4).

(4) [[x, disappeared, stark]]

The 'x' marks, as explained earlier, where to insert the subject, i.e. the topic. This replacement, where in this case 'x' becomes 'guardian', is then done in all of the information lists (here only one), in a predicate called `replace_xs/3`. The list that the KR module leaves as output, being the input to the sentence generation module, will look like (5).

(5) [[guardian, disappeared, stark]]

We have now shown two complete and proper examples of what happens to the player's input questions in the KR module. However, the input can be vary varied,

and the process is not always this simple and straightforward. Therefore, we will now show what happens when the player, including no possible topic or question word gives an undesired but possible input question, and that also includes a misspelled word.

Let say that the player's next question, after "Where is the guardian?" is "Why did it dissappear?" where 'it' is supposed to refer to the guardian and 'dissappear' is a misspell for 'disappear'. This kind of misspelling leads to QTAG not finding the word in its lexicon. Instead, QTAG tags the misspelled word by looking at its three last letters. In this case, where it is a verb that has been misspelled, the consequences do not make any harm since our inference machinery never treats verbs. But, if the user misspells a word that either was intended to be, or unintentional becomes a word that is treated by our program, it could lead to that the user does not receive an answer to the intended question, or an answer to the unintended question.

We will here start from the beginning and in detail explain what happens to misspelled input from the player, where also no reference can be made. The input to the KR module, given that the player's second question is "Why did it dissappear?", is (6), where the misspelled 'dissappear' has been given an arbitrary tag.

(6) [(why,wh), (did,v), (it,pro), (dissappear,ARB)]

Starting to look for a question word, we realise that none of the words tagged 'wh' is 'where' or 'who'. The current question word is therefore set to 'any'. The program then aims to define the topic of the input by first extracting the nominal heads, resulting in (7) where no direct match between the list of candidates and the keys of the KB can be found since 'it' is not a key in the KB.

(7) [it]

The program then tries to find a synonym to one of the candidates (here just one) that matches a KB key, using the WordNet Prolog database. It turns out that the word 'it' does not have a synonym that is a key in the KB.

The next possibility to define a topic, is that 'it', the only candidate, is an anaphoric reference to the previously defined topic, in this case 'guardian'. The program then checks in its lexicon if the features of the both words, 'guardian' and 'it', are compatible. Since 'guardian' is a noun with the features of third person singular and masculine, and 'it' is a personal pronoun that can only refer to a word with the features of third person singular and no specific sex, 'it' cannot refer to 'guardian'. This is because 'guardian' is a masculine word (in our KB lexicon) and the KB can only match this kind of word with the pronoun 'he'. Since no anaphoric reference between 'guardian' (the previous topic) and 'it' can be made, the topic is set to 'any'.

The program now continues to search the KB for information, but with an undefined question word and an undefined topic, an empty list of information is returned. The output from the KR module, and likewise input to the sentence generation module, is (8).

(8) [[]]

4.3 The Generation Module

The generation module consists of three different parts, the DCG, the lexicon and the generation system itself, which in turn consists of several different Prolog predicates. The grammar is quite simple, but since the sentences in the fragment of English that we are covering have very similar syntactic structure, there is no need to make the grammar more complex. The grammar and lexicon can be viewed in Appendix C. The lexicon is also very simple and we have only included a small number of features in the lexicon used in the generation module, such as features that control subject-verb agreement and determiner-noun agreement. The generation system itself will be described in detail in the next section. An overview is also presented in Figure 6 below:

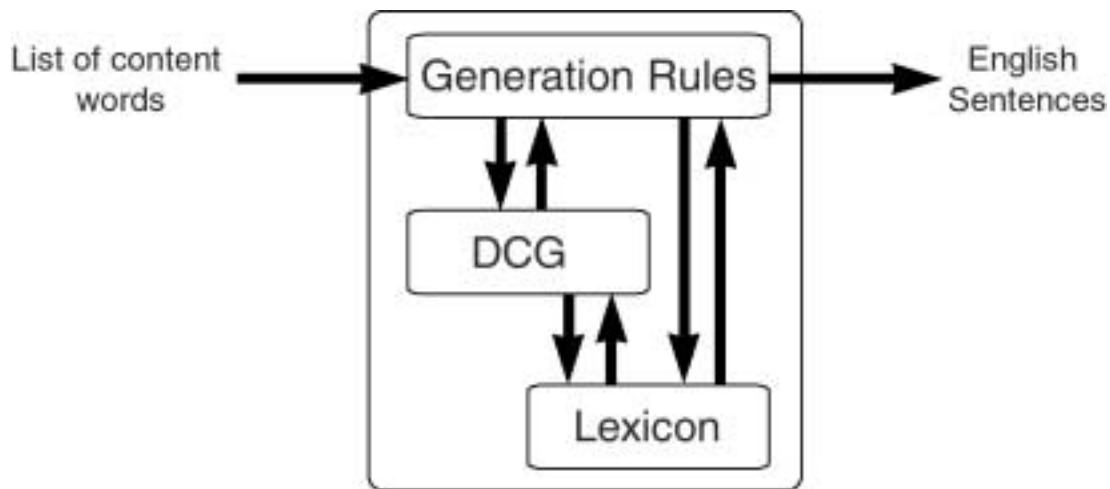


Figure 6: An overview of the Sentence Generation Module, showing flows between grammar and lexicon and the generation rules.

4.3.1 An Overview of the Generation Module

The generation system takes content word lists as input and makes syntactically correct sentences from the input words, starting from the end of the input word sequence. The reason for starting at the end of the word sequence, working backwards, is simply that we need to find the possible phrases that might already

exist in the input, before we work out which new words need to be added in the generation process. The generation method itself consists of two main procedures.

The first procedure, which we have chosen to call ‘abstracting’, makes a phrase out of one content word or a combination of several content words that appear in the input list. We felt that abstracting is an appropriate term for such a procedure, since it involves going one step higher in the bottom-up parsing. An example of abstracting is when a single proper noun is abstracted into a noun phrase, since we have a grammar rule that says that a noun phrase can be made up of a single proper noun. Abstracting is needed in the generation process, in order to group the different input words together in the different phrases that make up one sentence.

After the abstracting step has been carried out, the second main procedure comes in. The second procedure, which we have named ‘expanding’, means that we expand the input word sequence with additional words, which were not present in the input list given to the generation system. It is important that abstracting is done before we add new words to the word sequence, to prevent that unneeded words are inserted, or that needed words are inserted in the wrong position in the word sequence. The importance of the abstracting step is illustrated in the example below:

If we have the word sequence [parts,given,enchanted,people] and we start from the back with the word ‘people’, we recognise that this is a noun, and from this we want to create a noun phrase. To do this, we need to insert an additional determiner, to create the noun phrase. This can be carried out in the expanding procedure, and the word ‘the’ can be inserted in front of ‘people’, to make up the noun phrase ‘the people’.

After this expanding step, we would have the sequence [parts, given, enchanted, the, people], but since the word ‘enchanted’ was also supposed to have been included in the noun phrase, as a modifying adjective to the noun, the generation will now fail, since the determiner has been inserted in the wrong position and no syntactically correct sentence can be generated. This is why the abstracting step is needed, to group the different parts of speech together, before new words are inserted into the word sequence.

Our generation system is constructed in a way so that the minimum amount of extra words needed to make a syntactically correct sentence is inserted into the input word sequence. This is done to avoid adding words that were not intended to be generated. A grammar might, for example, allow multiple preposition phrases in one sentence, such as: “The guardian lives in the tower in Stark”. In the input word sequence [guardian,lives,tower], however, the second preposition phrase is not intended, since the last content word (Stark) is not present in the input sequence, and here we only want to generate “The guardian lives in the tower”. For this reason, it is important that the generation method used does not add more words than necessary to make up a sentence, since this would cause a loop in the program and structures such as “The

guardian lives in the tower in Stark in the tower in Stark...” etc. will be generated. The multiple preposition phrases will only be generated when there is a second content word to match the second preposition, such as in [guardian,lives,tower,stark]. Our grammar does not allow this kind of multiple structures, since we have simplified our sentences to limit the amount of grammar rules as well as the size of the lexicon, but in principle, our generation system should work on more complicated grammatical structures. This is because it only adds the smallest amount of extra words to every input word sequence, instead of looping and adding several extra phrases when they are not needed.

4.3.2 A Detailed Description of the Whole Generation System

We will here outline our whole generation system and describe every generation step in detail. The predicates can be found in Appendix D.

generate_all/2

The main generation predicate is called generate_all/2. This predicate finds all sentences that can be generated from a list of input words, applying the predicate generate/2 to the input words. The generate/2 predicate generates one sentence from the input words at a time, but since generate_all calls the predicate findall/3 and applies this to generate/2, all sentences that can be generated in generate/2 are found.

When all the sentences have been generated, there is a parsing step, which controls that all of the generated sentences are correct, according to the grammar. This step might seem unnecessary, since we should only generate correct sentences in the first place, but in the generation system, we have excluded the features, which are included in the grammar and lexicon, to make the generation process quicker. This means that a sentence like “The parts was given to the enchanted people” can actually be generated in generate/2, but this ill-formed sentence will be filtered out in the parsing step in generate_all/2.

It is only ill-formed sentences caused by the lack of agreement features that can be generated in generate/2. Syntactically ill-formed sentences, such as **“The parts given were to the enchanted people”* or **“The parts were given to enchanted the people”*, will never be generated, but since we do not check for agreement in the generation step, some sentences without agreement between subject and predicate will first be generated, but these sentences will then be eliminated in the parsing step, which is the last thing that happens in our main predicate generate_all/2.

generate/2

We will explain every step in this predicate and illustrate using an example. The `generate/2` predicate takes a list of input words (for example `[plate,key]`) and generates one sentence in the following way:

- 1 The input words are looked up in the lexicon, by the predicate `lookup/2`, which returns a list of the words respective part of speech tags, as well as the features of every word (person, number etc.). Output from `lookup/2` will here be: `[n,3,sg,n,3,sg]`.
- 2 The list of part of speech tags and features is sent to the predicate `remove_features/3`, which separates the words' features from their part of speech tags, and returns two lists containing them separately. Output from `remove_features/3` will here be: `[n,n]` and `[(3,sg),(3,sg)]`. The reason we made `remove_features` return the feature list, was that we first intended to check for agreement in the generation process, but at a later stage we decided to ignore agreement between subject and predicate and instead check this in a separate parsing step.
- 3 The part of speech list is then sent to the predicate `abstract_and_expand/3`, which we will describe in more detail later, but for now we can say that it first performs abstraction, i.e. tries to group the parts of speech together into new phrases, and then performs expansion, which is to find what is missing in order to make a phrase (and later a full sentence). Output from `abstract_and_expand` will in this first step be: the `ExpandedList` will be `[n,det,n]` and the `MissingCat` will be `'det'`. This means that what we should do next is to find a determiner and insert this and after that perform the generation on the new list. Note that there is actually no abstraction done in this first step, since `[n,n]` cannot be abstracted without additional categories inserted.
- 4 The next step in `generate/2` is an if-statement, where we check for correlation between the original part of speech list and the expanded list. This correlation check is necessary because what we are trying to achieve is to find the correct position in the original input word list `[plate,key]`, where we should insert the new determiner. In this case, we can check the position in `[n,det,n]` and insert the determiner in the same position in the original list, i.e. in the second position. This is because the noun in front of the determiner only consists of one word.

The correlation check might then seem unnecessary in this example, but at a later stage, we will have the following expanded list `[np,vbar,np]`, where the `vbar` stands for the word `'is'`. If we were to use the same method for finding the right position as in the previous stage, we would again get position 2, since the `vbar` is the second element in the expanded list. But here, the noun phrase correlates with two words, and we should actually insert the verb in the third

position (since we want to get “The plate is the key” and not **“The is plate the key”*”).

The if-statement checks for this kind of correlation and performs the right call to `nth/3` depending on if correlation is needed or not. The `corr/3` predicate correlates the original part of speech list with the expanded list and comes up with a correlation if there is one. If the head of the if-statement is true, the `nth/3` is performed on the correlated list, but if it is false, the `nth/3` is performed on the expanded list itself. The `nth/3` predicate is SICStus Prolog’s own built in predicate for finding the position of a certain element in a list. In the former case (when the expanded list is `[n,det,n]` and the missing category is ‘det’) we will perform the `nth-check` on the expanded list and get position 2 and in the latter case (when the expanded list is `[np,vbar,np]` and the correlated list is `[det,n,vbar,det,n]` and the missing category is `vbar`) we will perform the `nth-check` on the correlated list and get position 3.

- 5 The next step in the generation process is to find the actual missing word, which we should insert at the position just found, in the original word sequence. This is done by using the `findall/3` predicate on the predicate `apply/5`. The predicate `apply/5` takes the missing category and finds corresponding words in the lexicon to this category. In the first stage, where the missing category is ‘det’, we will in this step find all words in the lexicon that corresponds to this category, i.e. ‘the’, ‘a’, and many others.
- 6 We will then send the list of the possible missing words (determiners in our case) to the predicate `insert_to_inputlist/4`. This predicate takes the list of missing words and puts one at a time into the original word sequence, at the position that we produced in step 4. The output from `insert_to_inputlist` is a new list, where the first missing word is inserted. Here, it will be `[plate,the,key]`.
- 7 The next step is also an if-statement, where the head of the statement is a call to `abstract_and_expand/3`, to see if the next turn using this predicate will result in a full sentence. If so, we are already finished, and output from the `generate/2` predicate will be the new input list that we produced in the previous step. In our example, where we have `[plate,the,key]`, we are not finished, and the head of the if-statement will become false, and a recursive call to `generate/2` is made, but here we send the new input list as input, which means that the next turn in `generate/2` will be made on the input `[plate,the,key]` instead of just `[plate,key]`. After a few more laps in `generate/2`, we will have the correct sentence “The plate is the key” and the if-statement’s head will become true, and `generate/2` will stop, resulting in the full sentence.

Note that `generate/2` only produces one sentence. It is the `findall` call in `generate_all/2` that prompts `generate/2` to find all possible grammatical solutions for the original input word sequence.

abstract_and_expand/3

This predicate is an important one, since it is called upon by our main generation predicate. The first thing that happens when this predicate is called, is that the abstraction predicate, `check_all/2` is called. When no more abstraction can be made, i.e. when no phrase can be made from the current parts of speech without inserting a new word, `check_all/2` terminates and the expansion predicate, called `insert_all_links/3` is called. This predicate produces the first missing category and inserts it into the part of speech list.

We will describe both `check_all/2` and `insert_all_links/3` in detail later, but for now let us show an example of what comes in and goes out of a call to `abstract_and_expand/3`. The example is the same as above, and input to `abstract_and_expand/3` is the part of speech list `[n,n]`, which corresponds to the input words `[plate,key]`. When `check_all/2` is sent the `[n,n]` list, it will not be able to produce any phrases directly from these parts of speech (since there is no grammar rule that says a phrase can go directly to a single noun or to a combination of just two nouns). Therefore, nothing will happen in this stage and `check_all/2` will simply return the same list as output. The predicate `insert_all_links/3` will then receive the same list, `[n,n]`, and try to find the first missing category that is needed to produce a phrase, and insert this in the right position in the part of speech list. In this case, `insert_all_links/3` will find that a determiner is needed to make a noun phrase from the last noun in the list, and output from the predicate will be a new part of speech list, where the missing category 'det' is inserted in the correct position: `[n,det,n]`.

check_all/2

This predicate is a recursive one, thus needing a base case, which is the following: `check_all/2` terminates when there are no more phrases that can be made directly from the parts of speech in the list. This is tested in the predicate `no_more_phrases/2`, which calls on two other predicates, `transform/2` and `out_of_phrases/1`. The `transform/2` predicate checks if there is any category in the part of speech list that can make up a phrase on its own. This is the case with proper nouns or personal pronouns, since these categories on their own can be abstracted into noun phrases, without combining them with other categories. If such a transformation is found in `transform/2`, the rest of the list is evaluated in `out_of_phrases/1`. If a transformation cannot be found (such as in our case with `[n,n]`), the whole list is evaluated in `out_of_phrases/1`.

The predicate `out_of_phrases/1` is a kind of boolean predicate, which succeeds if it is impossible to make another abstraction in the part of speech list, i.e. if we cannot make any more phrases directly from the current categories.

Note that in the base case of `check_all/2`, the transformation predicate only tests if a transformation can be made. In the base case, in the end of `no_more_phrases/2`, the output list is always assigned the original input list. This is done because the real transformation is to be made in the next call of the predicate `check_all/2` and not in the base case. The base case should only test if there are no more abstractions that can be made. This means that if the part of speech list `[pn,n]` is sent to `check_all/2`, the base case will check `no_more_phrases/2` and come to the conclusion that there actually are more abstractions possible. Therefore, the base case will fail, and the next `check_all/2` will be called upon, using the same list as input, `[pn,n]`. In the previous example, however, we had `[n,n]` and this input list will exit in the base case of `check_all/2`, since it is true that no more phrases can be produced directly from these parts of speech.

The other `check_all/2`, which is the ‘real’ predicate (i.e. not a base case), first calls the transformation predicate described above, to make phrases out of categories that on their own can make up phrases. If no transformation can be made, the same list is returned from `transformation/2`. The output list is then sent to the predicate `check_next/2`, which is a predicate that only checks for one abstraction at a time.

`check_next/2`

This predicate is also a recursive one and it terminates when there are no more abstractions possible. To explain this predicate, we need a new example, since the previous one (`[n,n]`) terminated in the base case of `check_all/2` and this predicate never was called. Imagine the next stage in the generation, when we have found the first missing category and inserted this into the list. Here we would have the list `[n,det,n]` and this list would not terminate in the base case, since there is one possible abstraction to be made. The predicate `check_next/2` would be called, and this is what happens:

- 1 The last element in the list is removed by the predicate `delete_last/3`. The output is the removed element – ‘n’ – and a list without the last element present – `[n,det]`.
- 2 The second last element in the list is removed by the same predicate. The output is the removed element – ‘det’ – and a list without the second last element present – `[n]`.
- 3 If the elements that were just removed from the list can make up a phrase P, the phrase P is inserted in the list, in the position of the two removed elements. In this case, there is such a phrase, since the determiner and the noun can be abstracted into a noun phrase, ‘np’. This is inserted into the list – `[n,np]`. If the elements cannot make up any phrase, we instead call `check_next/2` recursively, this time using the list we received after removing the last element, as input. After the recursive call to `check_next/2`, we also put the last element back into the list. In our example, we will not get to the ‘else’-part of the if-statement, but consider an example where the following list would be sent to `check_next/2`: `[det,n,np]`. The last element ‘np’ would be deleted and the list `[det,n]` would be

received. Then the element ‘n’ would be deleted. The if-statement tries to make a phrase from the two categories, but since there is no grammar rule that says a phrase can consist of a noun followed by a noun phrase, the head of the if-statement fails, and we end up in the else-part. Here, we call `check_next/2` again, using the list `[det,n]`. This time, we will perform an abstraction in the if-statement, since there is a phrase (‘np’) to be made from the two categories. We insert this new phrase in the old elements’ position and get `[np]`. After this, we add the element removed in the previous step, ‘np’, to the list and receive `[np,np]`. Here, `check_next/2` terminates.

`insert_all_links/3`

This predicate is called only when there are no more abstractions to be made. The `insert_all_links/3` predicate is actually the essence of the expansion procedure mentioned above. It takes as input the list, in which there are no more possible abstractions, and produces the next category, that is missing to make the next possible phrase, and then inserts it into the list. In our first example, we had the list `[n,n]` and when this was sent to `check_all/2`, it terminated in the base case, since there were no possible abstractions to be made. The list is then sent to `insert_all_links/3` and this is what happens:

- 1 The last element in the list is removed by `delete_last/3`.
- 2 The second last element in the list is also removed by the same predicate.
- 3 The predicate `link/5` checks if there is any link between the two removed categories and finds the next missing category that is needed to make this link. In this case, `link` will produce the category ‘det’ and insert this into the list: `[det,n]`.
- 4 The built-in predicate `append/3` is called, attaching the remainder of the list, after the two elements were removed (`[n]`), and the list produced in `link/5` (`[det,n]`), producing the new list `[n,det,n]`.

We will conclude this detailed description of our generation system by presenting a full example, leaving out the transformations and only showing the actual word lists:

```
Input: [plate,key]
Abstracting: fails – [plate,key], [n,n]
Expanding: [plate,the,key]
Abstracting: succeeds – [plate,the,key], [n,np]
Abstracting: fails – [plate,the,key], [n,np]
Expanding: [plate,is,the,key]
Abstracting: succeeds – [plate,is,the,key], [n,vbar,np]
Abstracting: succeeds – [plate,is,the,key], [n,vp]
Abstracting: fails – [plate,is,the,key], [n,vp]
Expanding: [the,plate,is,the,key]
Abstracting: succeeds – [the,plate,is,the,key], [np,vp]
Abstracting: succeeds – [the,plate,is,the,key], [s]
Output: [the,plate,is,the,key]
```

Note that this is only one possibility of the input list [plate,key], as generated by generate/2. The main predicate generate_all/2 would return all of the possibilities, such as for example [the,plate,is,a,key], [the,plate,must,be,the,key] etc.

4.4 The Post-processing Module

As described earlier, we simplified the dialogue from the original game situation, to make it easier and more appropriate for our system. This means that we made the different sentences and structures simpler and more similar to each other, to save the number of grammar rules needed to generate all of the sentences. For this reason, our system only generates simple and short sentences and this leads to the output from the generation system sometimes looking static and boring. We therefore came up with an idea on how to make the output sentences look more varied and alive and implemented this in a post-processing module.

The post-processing module comes in after the generation process and makes a few different changes to the output from the generation module, for many different purposes. The post-processing consists of six steps, with different purposes and functions:

1. Deletion of incorrect verb-preposition phrases, such as **“The guardian lives at the tower.”*
2. Correction of ‘an’ vs. ‘a’, such as **“A enchanted plate was given to the people.”*
3. Deletion of wrongly generated numerals, such as *“Both parts were given to the people.”* (Although this is syntactically correct, the parts in question come from the plate being broken in four parts, hence the word ‘both’ is not semantically correct.)
4. Anaphor insertion, which is applied when two sentences that are generated after another share subject. An anaphor is then inserted in the second sentence, to make the text smoother. Here is an example: *“The plate is in Arcadia. The plate is the key.”* will be turned into: *“The plate is in Arcadia. It is the key.”*
5. Sentence conjoining, which is applied by insertion of ‘and’ between two sentences. Using the example in (4), we would here receive the sentence *“The plate is in Arcadia and it is the key.”*
6. Print the sentences in a readable way

We have implemented these steps in the post-processing module, which takes the output from the generation module and makes some simple changes to the text, to make it more varied and less boring. The first three steps in the post-processing module could have been implemented in earlier stages of our system, some in the grammar that the generation system uses and some in the lexicon. The three final

steps, however, could not have been implemented earlier in the other modules, since they require the fully generated sentences as input. An overview of the module is shown in Figure 7.

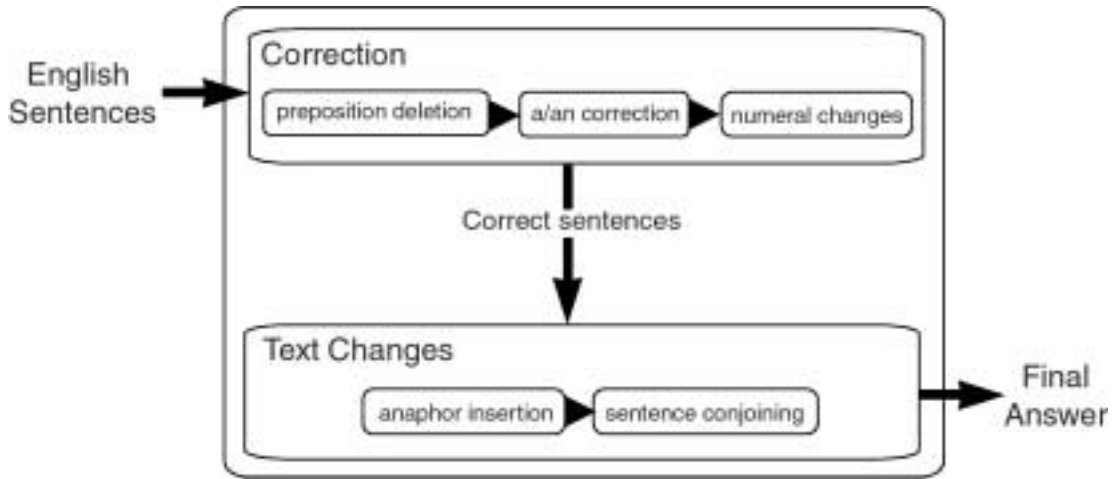


Figure 7: An overview of the Post-Processing Module, showing the different functions.

The anaphor insertion and sentence conjoining mechanisms, unlike the earlier three steps, do not have as a task to eliminate incorrect sentences, but to give an impression of textual awareness and to give a smoother text back to the user. This can only be done in a post-generating process, where more than one isolated sentence are considered. As mentioned, the first three steps of the post-processor would preferably have been dealt with in the grammar and lexicon, so that the incorrect sentences would never have been generated in the first place. However, since we did not realise these problems until after the full implementation of the generation system, we had to implement a separate module to handle these problems. If we could go back in time and do things differently, we most likely would have thought about these issues and dealt with them in the construction of the grammar and lexicon. More about alternative ways to deal with these problems later.

We will here outline the different steps in the post-processing module and later in this chapter go into detail about each of these steps. In the first step of our post-processor, we deal with verb phrases that have been given the wrong preposition phrase as a complement. Dougherty (1994) showed us how to force certain verbs to be followed by certain subclasses or prepositions, using the Prolog predicate `append/3`. Since we have chosen to implement our grammar as a DCG, we cannot follow his example.

In the early stages of our project we also did not think about the fact that the determiner 'a' should become 'an' in front of a vowel sound and even though it would have been an easy task to include not only 'a', but also 'an' in our lexicon, we

would still need some way of determining the vocal harmony in deciding their use. For this, the features in our current lexicon would not have been enough. The grammar in our generation system would even then generate incorrect nominal phrases, such as ‘an guardian’ or ‘an plate’ and a post-process to exclude all the wrong ones would still be necessary. Therefore, we have chosen to generate sentences only with ‘a’ and then correct it to ‘an’ where necessary, in a separate post-processing module.

The problems with the numerals could have been solved by not listing them as determiners in the generation lexicon. We currently make no difference between determiners such as ‘a’, ‘the’, ‘both’ or ‘much’ (apart from number, where ‘a’ has the feature singular and ‘both’ has the feature plural and ‘the’ has unspecified number) and the lack of distinction between the different determiners can lead to unintended generations such as “Both watchmen live in Arcadia.” when the desired generation was “The watchmen live in Arcadia”. Since we did not think about this distinction in the construction of the lexicon, we needed some way to correct these mistakes after hand and we have in this module implemented this so that the more specific numerals, such as ‘both’ and ‘much’ only occur in sentences where they were actually present in the output from the KB (i.e. they were already present in the input to the generation module).

We initially intended to implement an exchange between some of the generated words and their synonyms in WordNet, in order to increase the lexical variation of the output from the generation module. This would prove Danlos (1987) wrong, saying that knowledge represented by natural language cannot generate varied sentences. However, after testing WordNet in various ways, we found that the semantic network, with very detailed information and relations between all words, makes it impossible to implement such a synonym switch in a short time. All words in WordNet have many different senses, and it is impossible to know which of all senses is the intended one, without looking at the context of the word. Exchanging a word with an arbitrary sense with a synonym would many times give a sentence with a different meaning back. For example, according to WordNet, the noun ‘cat’ has seven different senses. If we were to replace the sense of a pet with the sense of ‘guy’ (which in one sense of the word is a synonym for ‘cat’), the output would definitely change in a way not intended. We will now in detail describe the implementation of the different functions in the post-processing module.

4.4.1 Prepositions

The first step is to exclude sentences that contain incorrect preposition phrases. The grammar does not include any semantic information, and therefore both “The guardian disappeared in Stark.” and **“The guardian disappeared about Stark.”* can be generated. This is because both the prepositions ‘in’ and ‘about’ are in the lexicon, where they have the same features. The incorrect forms are easily excluded, not from

being generated, but from being put to the user, in a predicate named `pass_correct_prep/2`. We have, in another predicate, listed the verbs in the lexicon and what prepositions they allow. A sentence including a verb followed by a preposition not in this list, is excluded and will not be shown to the user. This solution then demands a listing of all the verbs in the lexicon that can be followed by a preposition, but we have not come up with a better idea on how to put this exclusion in the grammar using our generation model. The listing is though easily done, and we found it accurate enough for our program. It is also very similar to Dougherty's solution (1994), in that he also listed what possible prepositions could follow a certain verb, though he did it within the grammar.

4.4.2 Indefinite Articles

In our generation lexicon, we have only included 'a' as an undetermined determiner, that is, we have not included 'an' which we see as a conjugated form of 'a'. Therefore, the grammar will generate both "a magical plate" and *"a enchanted people". In the post-process, we then have to check all these phrases and conjugate 'a' into 'an' where necessary. We have chosen to do this in a somewhat incomplete but yet satisfactory way, meaning that we do not concentrate on the sounds but on the letters. We say that when the determiner 'a' is followed by a vowel letter, it should be conjugated to 'an'. This incomplete solution would let "a hour" pass since 'h' is not a vowel and we have no way of checking the sounds of the letters. However, we still found that this ad hoc solution works well in our system. Our small lexicon also does not include any words where a non-vowel makes up a vowel sound, which means that these failures will never occur in our system.

4.4.3 Numerals

Another post-process that actually could have been handled in an earlier stage, by constructing the grammar in a different manner, is a way to handle different numerals. In our fragment, words like 'both', 'two' and 'much' occur. In the lexicon we define these as determiners, not making any difference between for example 'both' and 'the'. This is syntactically correct, they can often occur in the same environment, but semantically they differ. Realising the odd results from generation too late, distinguishing determiners from numerals would have demanded a lot of work. Therefore we have, as for the prepositions, implemented a post-process function for excluding the unwanted sentences from the list of possible output. A sentence with a numeral is not wanted if the numeral does not occur in the information stored in the KB, i.e. the input to the generation.

Let us look at an example: The player asks "Tell me about the plate", which after the input analysis and KB search results in a list of information, including the information [plate,opens,tower]. This information is the input to the generation

module and one of the generated English sentences is the in some way awkward *"Much plate opens the tower", since 'much' in the lexicon is defined as a determiner for a singular noun. The post-process will not allow this sentence to be given to the player since 'much' is listed as one of the numerals that cannot be allowed in a sentence originating from information that did not include it. Another sentence generated on the topic of plate and the KB information [plate,divided,four,parts], is "The plate was divided in four parts". Here, the occurrence of the numeral 'four' does not mean an exclusion of the word, since it occurred in the original information which means it is wanted in this case.

This function was also necessary looking from a broader view, namely on the textual level. When a player's question is answered with many sentences, occurrences of unwanted numerals could mean a semantically incorrect text. Without the implementation of this function, when asking general about the plate, the following output could be given to the player: "The plate is in Arcadia and it was divided in four parts. Both parts were given to an enchanted people.", where 'both' semantically cannot refer to 'four'. This will now not be a possibility; given the input [parts, given, enchanted, people] will not allow 'both' as a determiner.

4.4.4 Anaphors as Subjects

When the user asks something in general about a topic, for example the 'guardian', and the KB contains information that is generated in more than one sentence, these sentences often resemble each other in that almost all of them have the same subject, i.e. the topic. Asking, for example, "What do you know about the guardian?", some of the generated sentences will be "The guardian watches over the balance", "The guardian must return to the tower" and "The tower was built for the guardian", where the subjects of the two first sentences are the same, namely the guardian. Instead of showing these very similar sentences to the user, we compare two subsequent sentences, starting with the two first ones. If they share subject, i.e. if they start with the same prefix being a noun phrase, we change the latter to the corresponding anaphor. The lexicon shows that 'guardian' has the features of third person singular and masculine, which resembles the features of the personal pronoun 'he', and 'he' is inserted as the subject of sentence number two. We then continue to check the rest of the generated sentences, two at a time. In this example, the three sentences with information on the guardian will now be "The guardian watches over the balance", "He must return to the tower" and "The tower was built for the guardian".

4.4.5 Conjoining Sentences

We wanted to keep the grammar for generation as simple as possible to prevent looping. This means that the generated sentences sometimes are very short and to

give many short sentences following each other to the user, would make them look static. It is in the post-process easy to conjoin two subsequent sentences with an ‘and’ or a ‘but’, depending on the relation between the sentences. However, doing this without a semantic analysis of the sentences, it is impossible to say if they are equal or if one of them is subordinated to the other, as discovered in the TEXT project (McKeown 1985). We have therefore chosen only to implement the conjunction using the word ‘and’. We do this by simply taking two sentences at a time, and put an ‘and’ between them. The earlier example, asking “What do you know about the guardian?”, will now be answered with “The guardian watches over the balance and he must return to the tower. The tower was built for the guardian.”.

4.4.6 Printing Output

In order to present a more readable output, staying in the Emacs’ environment and without using any graphical interface, we have implemented our own print predicate. This is to reduce the pure Prolog output with for example list and variable notation and in order to give an impression of a textual output and not just raw programming output. This includes not printing a list of words as a list, but as a sentence with a space between the words, beginning a sentence with a capital letter and ending it with a ‘.’. Prolog treats all words only in small letters since they otherwise would be interpreted as Prolog variables. Therefore, the English personal pronoun for first person singular is treated as ‘i’. In our print predicate this is changed to ‘I’. Using the nicePrint/1 predicate now means that the list [i,do,not,know,anything,about,that] is put to the user as “I do not know anything about that.”.

4.5 Results

The resulting system flow can be viewed below, in Figure 8:

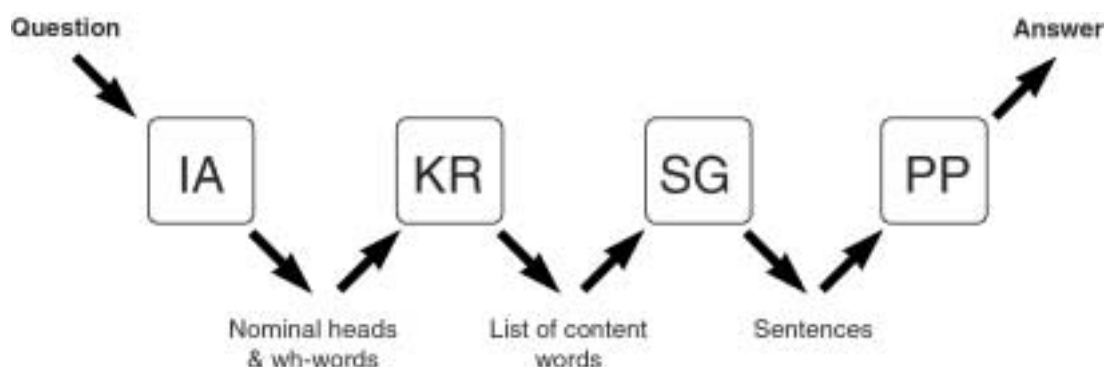


Figure 8: System flow overview, including the Post-Processing Module, which filters the generated sentences and chooses one.

We will here show examples of output from our system. We will begin by showing how our interface responds to the exact same questions as the ones in the real game. This dialogue will however be somewhat unnatural since the player, when typing the questions freely, most likely would not formulate the questions in the same way as in the game. We will still show what answers our system would give to these question, comparing the answers to each other and testing our input analysis.

Test run of our system using the same questions as in the real game:

> Hi, how can I help you?

I have to find the plate that unlocks the Guardian's tower

> The plate was divided in four parts and the parts were given to the enchanted people. The plate can restore a balance and it is the key. The plate opens the tower.

Where is the plate now?

> The plate is in arcadia.

Which people were given a part of the plate?

> The scriptures must tell about a people.

What scriptures?

> The scriptures can tell about an enchanted people and you can read the scriptures. Yerin protects the scriptures.

Where can I find these scriptures?

> The scriptures are in the library.

I also must find the entrance to the land of the Guardian.

> The entrance is in a tower.

I have to find two dragons that live in Arcadia.

> The dragons saved the human race and they contributed to the dichotomy.

How can I find out more about the dragons?

> The dragons saved the human race and they contributed to the dichotomy.

Where are the dragons?

> The scriptures can tell about the dragons.

Where can I find those scriptures?

> The scriptures are in the library.

Where is the library?

> A library is in the watchmens enclave.

We will now show another test run that demonstrates a more likely conversation where the player formulates her own question to the game character:

Hi!

Hi, how can I help you?

I'm looking for a disc.

> A plate was divided in four parts and the parts were given to the enchanted people. The plate can restore the balance and it is the key. The plate opens the tower.

aha, where is that tower?

> The tower was risen at a dichotomys creation.

I see, where was that dichotomy created then?

> The scriptures can tell about the dichotomy.

what scriptures?

> The scriptures will tell about the enchanted people and you must read the scriptures. Yerin protects the scriptures.

who is yerin?

> Yerin is the librarian.

where is the library?

> A library is in the watchmens enclave.

where is that?

> Did not get that, please rephrase.

where is the library?

> The library is in the watchmens enclave.

and where is that enclave?

> I do not know anything about that.

hm.

> Sorry, I cannot help you with that.

do you know who has the plate now?

> I have no idea.

what do you know about the enchanted people?

> The scriptures will tell about the people.

and the scriptures are in the library?

> The scriptures can tell about an enchanted people and you must read the scriptures.

is yering in the library?

> The library contains much useful information.

sorry, i meant: is Yerin in the library?

> Yerin is the watchman and you must talk to yerin.

where is he?

> Yerin is in the library.

okay. i might come back later. good bye.

> Good luck, I hope the information will help you to save our world.

The most striking difference between our interface and the one used in the game, is our short answers. Comparing our answers to the ones in the real game (see Appendix A), we can conclude that ours are shorter and that they respond more to the stated question. You only get to know what you ask for, and not much information on the background or history of the game or things. The lack of this kind of sentences can make the game boring, but on the other hand, you know that all information you get is needed. It is though not difficult to include some of this background information in our system. The content of this information can easily be added as content word lists in the KB. To achieve the syntactic structure of those sentences in the real game, more grammar rules in the DCG would also have to be included.

Every piece of information is not given when the player asks the exact same questions as in the game. For example no question about Yerin exists in the real game, which is likely to be the case if the player is allowed to ask any question at all. This proves that the difficulty of the game increases, since the player does not receive any extra information without having to ask for it, but explicit questions need to be asked in order to receive all of the important pieces of information.

As shown in the test run dialogue above, when the intended topic (Yerin) is misspelled (yering) and another nominal head (library) that is equivalent to a KB keyword appears later in the sentence, this nominal head is defined as the current topic. If a semantic analysis was performed, (and perhaps also a spell check,) the system would understand that what the user actually wanted to have information on,

actually was 'yering'. Our system cannot do this and, again, we note this as a restriction in our input analysis.

A drawback with our system, as it now is implemented, is that only exact conjugations are understood. One cannot ask for 'some plates' and retrieve information on the plate, as only the singular form 'plate' is stored in the KB.

When asking "where is that?" in the dialogue above, 'that' could not be understood as an anaphor for 'library', though this would be very easy to include in the lexicon. Where an answer consists of many sentences, it is quite obvious for the user that they are generated one at the time, even though we have implemented some post process functions that conjoin and infer anaphors for subject. If we were to implement a text generation system, the sentences would be more linked to each other, and for example, it would be possible to use anaphors not only for subjects but also for objects. This could also be done within our system if we would syntactically analyse the generated sentences.

Letting different people test our system, without having played the real game or knowing anything about the game situation, turned out to be impossible. Most of their questions had nothing to do with the game, though they tried to formulate questions on topics that often occur in adventure games. Since an answer is always given to every input question, the system still managed to proceed with the conversation.

Our interface and the real one differ in many ways, as do the conversations they allow. We have here only presented two different dialogues that are possible to have with our interface, but naturally many different ones are possible. The real game interface is really restricted and the different possible dialogues are very few. There is close to no variety and the simplicity of the game is striking: by clicking on lines on the display, you get to know everything you need to solve the puzzle. Since the lines are read aloud, you do not even have to be at your computer desk while playing the game. Sometimes the dialogues are so long, you can even go and do the dishes while playing it.

5 Discussion

In this chapter we will discuss the results of our overall achievements and also discuss particular parts of the system in greater detail and compare them to other similar projects.

5.1 The Input Analysis Module

It is difficult to make a relevant comparison between our input analysis to the one in the real game, since there actually is none to compare to. The recorded lines of the game all have hard-wired answers. We have not put any effort into constructing our input analysis so that those lines will be given the exact same answer. We found it unnecessary to do, as those recorded lines are not to expect if the player of the game is given the freedom to formulate her own questions. With a successful input analysis this should not be necessary to do, (see Section 4.5 for the results).

We early stated that we did not intend to create an understanding system. What we instead chose to do was to define what an input sentence is about, the topic, and what kind of question on this topic is requested. However, we conclude that our input analysis method, searching for nominal heads and wh-words, is insufficient. We rely on Clark's theory (1992) that introducing a topic is mostly done by mentioning a noun phrase. We do not solely search for noun phrases, but for the first appearing nominal head of the line that in some of the four ways corresponds to a keyword in the KB. Numerous examples could be shown proving this wrong, but looking at our results (see Section 4.5), we can conclude that for many various sentences, our analysis works.

We will now show an example comparing the real game's responses with our own system's answers. (The lines are taken from the original dialogue, see Appendix A.)

In the real game, the first question from the player and the corresponding answer are:

A1: I have to find the plate that unlocks the Guardian's tower.

T1: The plate that is the key? Yes, you will need it. It can even restore the Balance, if the new Guardian follows it to the tower, of course. Do you really want to find the plate on your own?

If the player were to ask our system the first line, A1, the answer given from the system would be S1. (This is actually just one of the many various syntactical structures that our system can give thanks to our sentence generation module, though all of them would have the same content.)

S1: The plate was divided in four parts and the parts were given to the enchanted people. The plate can restore a balance and it is the key. The plate opens the tower.

Looking at the third question in the original dialogue, A3, the corresponding answer, T3, and what our system would have responded, S3, we see that our system gives much less information when asked this question. This is due to the fact that our system is more precise when answering the question about the position of the plate, not giving the extra information about the story behind the plate. This information could still be received, if the player had asked a general question about the plate, instead of asking where it currently is (by including the ‘where’ keyword in the question).

A3: Where is the plate now?

T3: It was first stored out in the open, in the enclave of the watchmen in Marcuria, but when thieves tried to steal it, it was removed and split up into four parts. Each part was given to one of the four enchanted people in Stark – people who didn’t have anything to gain from the disturbance of the Balance.

S3: The plate is in Arcadia¹⁰.

If the player had asked a general question about the plate, such as “Tell me about the plate.”, the information given from our system would be something like S1. This information is very similar to the one given in T3, but it is actually not an answer to the question about the placement of the plate.

For those questions where our system’s answer is not as extensive as in the game, we would like to remind you that all essential information, necessary to have playing the game, is given. As our system only gives as much information as asked for, not for example answering the question *where* if not explicitly asked, our answers are sometimes shorter and the game more difficult. Having these results, we conclude that our input analysis module does not only cope with the questions of the real game, but it also enables the player to express herself freely, asking any questions in any order. This is a definite step forward from today’s adventure games.

5.2 The Sentence Generation Module

If we compare our sentence generation module to what we earlier described as Dale and Reiter’s language generation tasks (see 2.5.5), we see many differences in the division into sub processes, though resemblance in the content of them.

¹⁰ Marcuria is a town in Arcadia.

Dale and Reiter's first step is to decide the content of the output, what they call the message. In our system, the inference machinery in the KR module, choosing between general or specific questions, does this. Their next steps are to structure the messages and group them into sentences. We include these steps in our KB, where the information is stored and already grouped into sentences. The fourth step, lexicalisation, is also within the KB, where the information is realised as English content words. When it comes to Dale and Reiter's steps five and six, referring expression generation and linguistic realisation, we do these in reversed order. We first apply the grammatical rules and produce grammatical sentences. The sentences are parsed and corrected in different post-processes. The last thing we do, this also in a post-process, is to infer anaphoric references, looking at the close context. In a NLG system, which is what Dale and Reiter discusses, where a whole text is generated, these two last steps can be made in their order. But since we generate isolated sentences, and then put them together, we have to do the referring expression generation last.

As we see it, all of the six steps defined by Dale and Reiter, exist in our system. But since our sentence generation only is a part of a whole system, some of the steps can be carried out elsewhere. Many of their steps are therefore carried out in our KR module. This too, as we commented earlier in Section 3.4.1, shows that the knowledge representation one chooses is closely related to the generation of sentences or text.

Comparing our sentence generation module to the Compansion Project instead, we see that the input is the same, namely content words. However, both systems are open for other types of words, i.e. our generator sometimes receives prepositions and the users of the Compansion Project's system might type any kind of word. There is a difference in the input since in our system, the input comes from the KB which means that we can control it, where as the Compansion Project has implemented a module for unknown input. This means that their lexicon can learn new words and grow. Their lexicon is very complex as they define both features and possible functions for all entries, and so far, it covers only a small domain. For us, it is easier to expand the domain since we do not include syntax or semantics in our lexicon. Our system could perfectly generate "The green temper sleeps anxious.", a sentence that probably never could be generated by the Compansion Project's system. Our generation module trusts the KB not to include such information. There is also a difference in the output. The Compansion Project's system leaves it to the user to choose which one of the generated sentences that was the intended. Our system, after generation, parsing and post correction, randomly selects one of the possible sentences. This randomness was one of the main goals for our thesis. Next time, with the same input to the generation module, another grammatically correct sentence, though with the same content, will be put to the user.

It is obvious that the differences in the output depend on the different purposes for the two systems. Of course, we would have liked to include some semantics in our

lexicon, making it impossible to generate semantically ill-formed output, but as it is now, thanks to the composition of the KB, this is not a problem. And building our type of lexicon is so easy, we could easily implement a new one used for some other character in the same game, a possibility worth compromising for.

5.3 The Lack of a Discourse Module

We did not implement the mixed initiative that Walker and Whittaker (1990) brought up in their research. This is because we did not have enough time to implement the discourse module that would have been necessary to keep track of the shifts of control in the dialogue, over the different discourse segments.

In our system, it is only the player who can take initiative in the conversation as the system waits for input. The system itself cannot interrupt, start a new discussion or lead the player's questions. There is only one situation in which the system leads the conversation: when the treatment of the player's input starts the hypernym search for identifying topic, and the system keeps its turn by printing "Hm, wait a minute...". It is also impossible for the player to interrupt, as perhaps humans would say "Don't bother, it is not that important", but she has to wait for the system's complete answer.

A simple way to implement a kind of mixed initiative in our system would be to collect all possible topics in the player's input and ask the player which of them she wants to know more about, instead of choosing one noun as the first possible topic. This would, however, increase the time needed for the input analysis. As the system works now, it only detects one topic and answers with all information available on this particular topic.

Another drawback with the approach of asking the player which of the nouns she intended as topic (i.e. the noun that the player is asking the question about), is that extra turns are needed in the conversation. Many times these extra turns will most likely be unnecessary, since it might be obvious which noun in the input that is the topic. Look at the following example. Note that it is only a theoretical example and that our system would not respond in this manner.

Player: Where is Arcadia, the world of magic?

System: Do you want to know more about Arcadia, or about a world or about magic?

Here, using the approach of asking the player which noun should count as topic seems rather stupid. Since we already have limited ourselves in the input analysis, defining the first introduced noun phrase that matches knowledge in the KB as topic, is a good enough solution. We do not think that the system's impression on the player would improve by adding extra turns, asking the player to be more precise about what kind of information she wants. As mentioned in previous chapters,

defining the first introduced noun phrase as the topic of a single sentence goes along with Clark's conversational model of presentation-acceptance, as described in Section 2.6.1.

Walker and Whittaker states that when one chooses not to implement a discourse module, the system will not be seen as a conversation partner but as a question answering system. They also claim that anaphors in question answering systems often are treated as referring to the last question when it actually might refer to the last answer. We have solved this problem in our system by comparing the features of the anaphor and the current topic. If they do not match, the anaphor cannot be understood, meaning that there will be no incorrect references, but also no possibility to refer to anything else but the last discussed topic.

An example of this is the following dialogue:

- (1) Player: Tell me about the tower.
- (2) System: The tower was built for the Guardian.
- (3) Player: Who is the Guardian?
- (4) System: The Guardian protects the Balance.
- (5) Player: What is it?

Here, the last utterance will not be understood by the system, since the current topic is defined as 'Guardian'. The features of the anaphor 'it' are not compatible with the noun's features, since the gender feature for 'Guardian' is 'masculine, which would correspond to the anaphoric use of the pronoun 'he', not 'it'. This means that the player cannot refer to human characters in the game by using the anaphor 'it'.

However, the player is not referring to the Guardian, but to a noun introduced in the system's answer. This kind of anaphoric reference cannot be dealt with by our system. We only define topics introduced by the player, which is actually a result of us not implementing mixed initiative, which in turn comes from the lack of a discourse module.

Only letting the player introduce topics is not all negative. Bearing Clark's conversational model in mind, the player introduces the topic by uttering a noun phrase, such as in (3). The system makes the acceptance of the topic by letting the next contribution begin, i.e. by answering the question (since it recognises the topic from the knowledge in the KB and knows how to answer that particular question). It seems that Clark would agree with us here, that (4) does not introduce a new topic. The following question in (5) will therefore not be understood by the system, since the anaphor used does not refer to the latest introduced topic.

A discourse module would not only make it possible to implement mixed initiative and barge-in, but also the ability to remember what topics and information have already been discussed, much like Peedy in the Persona Project could remember

earlier discussions, by the use of a dialogue manager. If the player then gets stuck and does not know what to ask next, giving no input for a long time, the discourse module could allow the system to bring up a topic not yet discussed and introduce it to the player, perhaps by asking “Are you the one looking for the tower?”. If the system memorises what topics have already been discussed and a player asks a question twice, the system could note this, as in the real game where Minstrum Tobias repeats an answer by adding “Again, ...”. Memorising topics would also enhance the use of anaphors, making it possible for the player to refer to earlier discussions.

The lack of a discourse module is a major drawback in our game interface and it would have been impossible to implement our system as an agent without a discourse module. However, this limitation was a necessary one to make, in order for us to make a complete system within the scope of this thesis.

5.4 Modularity

Throughout the design and implementation of our system, we have tried to keep the modules of input analysis, knowledge representation, sentence generation and finally post-processing, separated from each other. The modularity of our system is high, and the four modules could easily be edited or replaced, which enables much further development. In this section, we will describe the modularity and show why it enhances our system.

The input analysis of our system does not make any semantic interpretations of the input words given in the player’s questions. It merely searches the input list for nominal heads and wh-words. One might therefore wish for a deeper analysis, preferably a semantic one to actually understand what the player asks. We will not say an analysis like that would be easy to implement, but we claim that it would be easy to expand or replace our current input analysis module with it. The input will have to be a list of words, if not to exchange the input reader function, and the output a list of possible topics and some sort of tag for specified knowledge. This is because the KR module needs this tagged list as input. If any other changes would be done, the KR would have to be changed in the same way.

The design of our knowledge representation module is not easily changed – the choice of knowledge representation is essential to the whole system, but the knowledge inside the KB, i.e. the content word lists, could easily be replaced storing for example another character’s knowledge in the game. The input to the KR has to be a list of possible topics and a tag that indicates what kind of information on the topic that the player has asked for. The output will have to be a list of content words, in any language depending on what generation grammar is chosen to implement in the sentence generation module.

We have not stored any knowledge on the system itself, which means that the player will not receive any information on the informative character himself (Minstrum Tobias). This is mainly because no information on Minstrum Tobias was given in the original dialogue in "Den längsta resan". In Julia (Foner 1997), this kind of knowledge turned out to be her Achilles' heal, giving away the secret of her implementation, but it also enabled her to answer many different questions about herself, for example about her age, name, mode and current feelings. August (Gustafson et al., 1998) has more than one KB, storing different kinds of knowledge into different files. To do this, a search for specific words in the user's input must be done, in order to know in which KB to search for the information. Having more than one KB is also possible in our system, which means that new characters with different knowledge could easily be implemented, just by adding a separate KB for that character. Naturally, any new words must also be added into the lexicon.

Eliza (Hutchison, 1986) exists in different versions with knowledge of certain domains, where the psychotherapist is the most famous one. The domains are though clearly separated and it is not possible to ask Eliza the psychotherapist about, for example, ticket prices. In the beginning of our work we actually contemplated implementing some knowledge on the system's character itself, but in the chosen scene of the game, no questions were about Minstrum Tobias and therefore we left this part out. Looking at August and Eliza, we would have done this with separated KBs, probably searching the input string for clusters of words to decide if it is a question on the character or on the game.

A grammar, a lexicon and a set of generation rules mainly build up the sentence generation module. These parts can be improved or exchanged, perhaps by including more features in the lexicon making some functions of the fourth module of the system, the post-process, unnecessary. When translating the game into another language, the grammar and lexicon have to be exchanged and we must stress the simplicity of doing this. As the lexicon now does not include many specified features, words could be added or the whole lexicon could be replaced. In order for our system to create varied output, not answering a question in the same way every time, it is essential for the generation module that its output consists of more than one realisation of the input content word list. There are many ways to realise the input, and as we randomly chose one of them as the answer, the output gets varied.

The post-processing module includes three different kinds of functions: correction, textual functions and printing. The correction functions, wasting the unwanted generated sentences and correcting some of the ill-formed sentences, are undesired but necessary. We would have preferred, as discussed earlier, that incorrect sentences were never generated in the first place. To achieve this, we would have to expand the lexicon and grammar, but no changes in any of the other modules would be necessary. The correction function (replacing 'a' for 'an') could also be invoked in the lexicon and grammar, demanding some work in the field of phonetics, as all

lexicon entries would have to be tagged with their initial sound. The textual post-processes cannot be invoked in any other module since its input has to be full sentences. This goes for the printing function too, as it prints a textual sentence, i.e. a conjoined sentence or a sentence with an anaphoric subject referring to the previous one.

5.5 Contributions and Limitations

We can see a number of contributions in our work on this thesis, especially in bringing linguistics into adventure games. Although we have not made the leap from the menu based conversational environment to the completely free chat-like interface in full, we have taken a few steps in the right direction. While “Den längsta resan” provides the player with a fixed number of possible questions, we allow the player to ask whatever questions she pleases, in any desired way. And while the game has pre-determined replies to each of the possible questions, we generate replies one by one, using knowledge that is stored separately, and the output becomes varied.

Making use of both QTAG and WordNet’s Prolog database file, the possibilities of our system increases, without needing to have large lexicons or KBs. The contributions that QTAG and WordNet have given to our system, would have been impossible, in the scope of this thesis, for us to implement ourselves. As discussed earlier, for recognising not only word forms in the KB or WordNet, a morphological analysis would be required. For this, we could have made use of an existing program if found.

Naturally, our system has its limitations. We would have liked to vary the output answers even more, perhaps by paraphrasing some of the answers that have been given to the player more than once. We would also very much have liked to use WordNet for lexical variation of the output and perhaps exchange one or several words in an answer for a synonym. This has not been done, as we have only used WordNet in the input analysis.

The main reason for us not using WordNet for lexical variation of the output, is that in order to exchange a word with its synonym, we need to know which sense of the word is the intended one. To find this out is a very difficult task. If we, for example, would try to exchange the word ‘guardian’ with a synonym in the sentence “The tower was built for the guardian”, we could, using the correct sense of the word, get “The tower was built for the protector”, which would be a good sentence. However, using another sense of the word, we might end up with “The tower was built for the steward”, which is not as good, since it will most likely not be understood by the player, even though in one sense of the word, ‘guardian’ and ‘steward’ are synonyms. Without analysing the context of a word, it is impossible to know which sense is the intended one and to guess or randomly pick one sense would cause too

much confusion since all of the words have many different senses. Using one frequent term from the game as an example, the word 'plate' has 13 different senses!

Our goal to produce varied output is to some extent also prevented by our small lexicon (which could easily be extended) and by our simple grammar: the shorter sentences we generate, the harder it is to syntactically vary them. As it is now, when the grammar can only produce easily combined sentences, the variation mostly lies in determiners and prepositions. We still feel that we have contributed to the field of text generation, since that not much previous research has been done on generation of single sentences, using as simple means as we have done.

Another way, in which we could have varied the output even more, would have been to enlarge the KB and add more knowledge to it. Since we need to keep the semantics of the game situation intact, we cannot add completely new knowledge, but we could add paraphrases to the knowledge already stored in the KB and then randomise which of the phrases are given to the player, when asked the same question twice.

For example, the KB has the information [scriptures,library] , which is an answer to where the scriptures can be found, and we could easily have added knowledge to paraphrase this into, for example, "You will find the scriptures in the library" or "The library has the scriptures.", by adding the corresponding content word lists for these sentences. However, since our interface is more an example of an idea on how to improve adventure game interactions, we have kept the KB's size small. Still, if there would have been a way of paraphrasing the output sentences, without us having to implement such a function ourselves, we would have liked to vary the answers as much as possible.

Our input analysis is not exhaustive, lacking for example a semantic interpretation to really understand the player's input. We see this part of our work as the greatest limitation. However, we have used the theories available for identifying the topic of a single sentence, concentrating on nominal heads. If we had performed a morphological analysis of the input words, forms not in WordNet or in the KB could also be understood. Still, WordNet is perhaps the greatest asset of our system since the storage of knowledge can be kept so small, even though the system understands so much more by the use of the semantic relations between the thousands of words that exist in WordNet. If we look at the amount of bytes we have used for writing the KB (2 kB), compared to the size of the WordNet Prolog database (8 MB), we can conclude that the possible use of our KB is much larger than the size of the hard wired KB.

6 Conclusion

One of the main goals of our work was to show an example of how to make the dialogues in adventure games more varied, not always answering the same question exactly in the same way. The game character's lines in the real game, which are recorded and read aloud, are of course vivid and complex, but they are still always the same, every time the player asks a particular question. The lines available for the player do not provide much freedom of choice since they are also always the same when encountering one specific character. In order to create a more diverse and flexible dialogue and game, we wanted to change these repeated conversations.

We have implemented an alternative language interface to one of the game characters in the game and although this interface is not actually integrated as a part in the real game "Den längsta resan", it functions as an example on how to improve the diversity in game dialogues. Our interface also enables more interaction between the player and the game character, which makes the game more interesting and fun. Our interface not only enhances the player's game situation, in that the player can ask any questions in any chosen syntactic form, but it also improves the static and boring way in which replies are given to the player. In the real game, the answers to the pre-programmed questions are hard-wired, whereas with our interface, by the use of text generation, we have added randomness and variation to the game character's answers. This was also one of the goals of our work.

We had no intention of building a grammar that is able to generate the exact same sentences read in the real game. Instead, we wanted to show that the same information could be split up into smaller fragments and presented in different ways, and when putting all of them together, the same information would be given by our system as in the real game. This is what we have done. We have created an interface where the player of the game may express her questions, as she wants, without restrictions. Letters can be capital or small, sentences can be without punctuation or even grammatically incorrect and the exact terms do not have to be used thanks to the use of WordNet. This, along with the fact that our system will always give an answer to any and all input question, makes our system very robust.

We put no effort into analysing the player's possible questions in the real game – those recorded lines were not considered likely to ever being used by the player if she had a choice to express herself. The freedom of expressiveness increases the difficulty and the feeling of interaction in the game. The answers generated in our system give as much information as in the real game – but only if you know what to ask for, which you should do if you have paid attention throughout playing the game. We believe that you are much less likely to pay this attention to the more controlled game dialogues, since they make you feel that you are not in control of the interaction but that it is your game character April Ryan who is playing the game for

you. Using our interface, the player becomes more involved in the game, since she needs to remember important information, in order to ask the right questions in the encounters with other game characters. We claim that our interface increases the difficulty of the game, but mean that this is actually a good thing, since it also enhances the feeling of the game, when the player becomes more active and involved in the adventure. And remember, it was the authors who made the claim that the brain is the player's greatest asset in solving the game.

7 Further work

We would have liked to implement a discourse model, even if just a simple one. An initial attempt to do this could be to solve user input that contains many possible topics in another way. Let us say that we save all of the possible topics in a list. The easiest way would then be to generate all information on all of these topics, giving all the information to the player at once. Or, to implement idea of mixed initiative, when the list contains more than one possible topics, the system has to ask which of the topics the user currently is most interested in, and then generate information on this particular topic.

Our system has no memory of what questions the user already has asked, meaning that unlike the real game, when a particular question is asked the second time by the same player, our system cannot note this. In the real game, Minstrum Tobias says, "Again, you will find the scriptures in..." when asked the same thing twice. It would not have been hard for us to do this, saving all of the system's output in a list and before answering a new question, check if this information already has been given.

If a player asks our game character a question, which is not understood by our interface, our system cannot deal with this by asking follow-up questions and getting the player to clarify herself. Our system merely states that it cannot help the player with that question, since it was not understood. This goes against Clark's model of collaboration (1992), which told us that when the listener (in this case the game character) cannot accept the speaker's contribution, the listener must ask implicit or explicit questions until the listener finally accepts the presentation. Again, by the use of a discourse model, this problem might be overcome and collaboration might have been achieved, which we think would have been very interesting.

Our input analysis model is, as noted earlier, not exhaustive. To understand the input, a semantic translation of it has to be made and compared to the content of the KB. We would also like to improve our system's input analysis with the understanding of conjugated forms. As it is now, the identification of topic only succeeds if the player writes the word of the same form as in the KB. And when using WordNet for topic identification, no conjugated forms are recognised. Therefore, we would like to implement a morphological analyser, recognising the input words' lemma and only store these lemmas in the KB and search for them in WordNet.

When translating the input into semantics, the form of the KB probably would have to change containing some sort of semantic representation in order to be able to compare the input to the knowledge. This would imply a whole different system than the one we have chosen to implement. To improve the existing one instead, we could tag the knowledge in the KB with more than the two current tags 'where' and 'who'. The input would then be scanned for further wh-words and the answers to the questions could be more specific.

To vary the output, more grammatical rules could be added, though this would certainly mean more ill-formed sentences if extra features in the lexicon were not also added. A different kind of output generation would be to paraphrase or exchange words for synonyms in the generated sentences. To do this successfully, a semantic representation is probably needed, as we discussed earlier. It would be interesting to use WordNet for lexical variation of the output words, but this means also analysing the word's context, to conclude which sense the word has, that is intended for lexical variation, in order to exchange that with a corresponding synonym. As mentioned earlier, adding a paraphrasing component to our post-processing module, would also have been an interesting way to vary the output from our system even more.

The ultimate goal for us would be to fully integrate our system into the real adventure game and to actually test it in the intended environment. However, for this to work, the need for speech recognition and synthesis must be met, in order to give the player the opportunity to talk to the game, and for the game to read aloud the generated sentences.

8 Summary

One of the goals for this thesis was to bring linguistic knowledge into the world of adventure games and by using input analysis and sentence generation create a more vivid game experience and varied output in game dialogues.

We have used a POS tagger to analyse the input, we have enabled topic identification in terms of synonyms, anaphors and hypernyms, and we have used a grammar and a parser to produce English output. The difference between earlier agents, such as the pioneering Eliza and the modern Julia, and our agent-like interface, is basically the use of linguistics. As they match input to hard-wired output, we analyse and generate.

Our input analysis is simple though efficient as it enables free input from the player without restrictions on syntax or semantics. The information on the game characters is stored in a knowledge base where information easily can be added or replaced in order to model more than one character of the game. The size of the knowledge base is kept small as we use WordNet, a great asset when creating linguistic software where the connection between words is crucial. The sentence generation module prevents the dialogues from always being the same, as it for each question from the player is able to generate a whole new answer, though always with the correct content.

We have shown that the user of our interface does not have to write grammatically or syntactically correct sentences, use punctuation or even the specific words used in the rest of the game. We think that even creators of adventure games should give the dialogues and the linguistics of the games some priority and to use this kind of knowledge to improve the overall impression of their games. That, along with the fact that the player can ask the questions in whatever manner she likes, is our interface's greatest achievement. We think that the simplicity of our generated sentences is compensated by the variety of dialogues and output possible to have. Using our kind of interface instead of today's hard-wired lines would make adventure games more adventurous, interesting, difficult and interactive.

References

- Allen, J., *Natural Language Understanding*, The Benjamin/Cummings Publishing Company Inc., 1995
- Allen J., Perrault, C., *Analyzing Intention in Utterances*, *Artificial Intelligence*, 15(3): 143-178, 1980
- Allwood, J., Andersson, L-G., *Semantik (Guling 1)*, Institutionen för Lingvistik, Göteborgs universitet, Kompendiet, Göteborg, 1996
- Ball, G., Kurlander, D., Ling, D., Miller, J., Pugh, D., Skelly, T., Stankosky, A., Thiel, D., Van Dantzich, M., Wax, T., *Lifelike Computer Characters: The Persona Project at Microsoft Research*, *Software Agents*, The MIT Press, 1997
- Blackburn, P., Bos, J., *Representation and Interface for Natural Language: A First Course in Computational Semantics*, 1996
- Brachman, R., Fikes, R., Levesque, H., *KRYPTON: A Functional Approach to Knowledge Representation*, *IEEE Computer* volume 16(no10), 1983
- Bradshaw, J.M, *An Introduction to Software Agents*, *Software Agents*, The MIT Press, 1997
- Clark, H., *Arenas of Language Use*, The University of Chicago Press, 1992
- Dale, R., Reiter E., *Building Applied Natural Language Generation Systems*, in *Journal of Natural Language Engineering* 3:57-87, Cambridge University Press, 1995.
- Demasco et al., *Towards More Intelligent AAC Interfaces: The Use of Natural Language Processing*, Demasco, McCoy, Gong, Pennington and Rowe, 1989
- Demasco, P., and McCoy, K., *Generating Text from Compressed Input: An Intelligent Interface for People with Severe Motor Impairments*, *Communications of the ACM*, May 1992, Vol. 35, No 5
- Dougherty, R., *Natural Language Computing, an English Generative Grammar in Prolog*, second edition, Lawrence Erlbaum Associates, Inc., Publishers, New Jersey, 1994
- Foner, L., *Entertaining Agents: A Sociological Case Study*, MIT Media Lab, The first international conference on autonomous agents (Agents '97), Marina del Rey, CA, 1997

- Gustafson, Lindberg, Lundeberg, *The August Spoken Dialogue System*, Centre for Speech Technology, KTH, 1998
- Hovy, E., Lin, C-E., *Automated Text Summarization in SUMMARIST*, In: *Advances in Automatic Text Summarization*, I. Mani and M. Maybury (editors), 1999
- Hutchison, C., http://www.cogs.susx.ac.uk/local/teach/ct_eliza, 1986
- Jones et al., *Knowledge Representation Considerations for a Domain Independent Semantic Parser*, Jones, Demasco, McCoy and Pennington, 1991
- Lager, T., *Semantik över satsgränserna - en samling diskurssemantiska miniatyrer*, STTS Report No. 90-10, Department of Philosophy, Göteborg University, 1990
- Levinson, S., *Pragmatics*, Cambridge University Press, 1993
- Mason, O., Personal Communication (e-mail), 2000
- McCoy et al, *A Semantic Parser for Understanding Ill-formed Input*, McCoy, Demasco, Gong, Pennington and Rowe, 1989
- McCoy et al, *A Domain Independent Semantic Parser for Companionship*, McCoy, Demasco, Jones, Pennington and Rowe, 1990
- McKeown K., *Text Generation*, Cambridge University Press, 1985
- Peregrin, J., *Topic, Focus and the Logic of Language*, Sprachtheoretische Grundlagen für die Computerlinguistik, Proceeding of the Göttingen Focus Workshop, 17.DGfs, IBM Deutschland, 1995, available at <http://www.gacr.cz/peregrin/HTMLTxt/tflog.htm>
- Reiter, R., *On Reasoning by Default*, Proc. TINLAP-2, Theoretical Issues in Natural Language Processing - 2, University of Illinois of Urbana-Champaign, 1978
- Shoham, Y., *An Overview of Agent-oriented Programming*, In *Software Agents*, ed J.M. Bradshaw, Menlo Park, California: AAAI Press, 1997
- Stede, M., *Lexical Semantics and Knowledge Representation in Multilingual Text Generation*, Kluwer Academic Publishers, 1999
- Walker, M., Whittaker, S., *Mixed Initiative in Dialogue: An Investigation into Discourse Segmentation*, In *Proceedings of the 28th Annual Meeting of the ACL*, 1990

(WWW1): *QTAG: The Birmingham-Lancaster Tagset*
<http://www-clg.bham.ac.uk/oliver/java/qtag/BLT-tagset.html>

(WWW2): *QTAG: A portable Parts of Speech Tagger*
<http://www-clg.bham.ac.uk/oliver/java/qtag/>

(WWW3): *Eliza as a JAVA applet*
<http://www.eki.mdh.se/kursshemsidor/informatik/team1/capron/CapWeb/yippee/yipsce/eliza/eliza.htm>

(WWW4): *WordNet: An Electronic Lexical Database*
<http://www.cogsci.princeton.edu/~wn/>

1 INTRODUCTION	1
2 SCIENTIFIC BACKGROUND.....	3
2.1 AGENTS.....	3
2.2 EARLIER PROJECTS	4
2.2.1 <i>The Persona Project at Microsoft</i>	4
2.2.2 <i>Eliza</i>	6
2.2.3 <i>The August Spoken Dialogue System</i>	7
2.2.4 <i>Julia</i>	7
2.3 INPUT ANALYSIS THEORY	9
2.3.1 <i>Identifying Topic</i>	9
2.3.2 <i>Formalising the Intentions and Plan of a Dialogue</i>	10
2.4 KNOWLEDGE REPRESENTATION.....	11
2.5 TEXT GENERATION	12
2.5.1 <i>TEXT</i>	12
2.5.2 <i>The Companions Project</i>	13
2.5.3 <i>SUMMARIST</i>	15
2.5.4 <i>On Verbs and Prepositions</i>	15
2.5.5 <i>Language Generation Tasks</i>	16
2.6 DISCOURSE	17
2.6.1 <i>Discourse Structure and Turn-taking</i>	17
2.6.2 <i>Discourse, Collaboration and Mixed Initiative</i>	18
2.7 SUMMARY	21
3 METHODS	23
3.1 OVERVIEW OF THE SYSTEM	23
3.2 TOOLS AND ASSETS	25
3.2.1 <i>The Adventure Game</i>	25
3.2.2 <i>WordNet</i>	28
3.2.3 <i>QTAG</i>	31
3.3 INPUT ANALYSIS METHOD	32
3.3.1 <i>Definition of Topic</i>	33
3.3.2 <i>Distinguishing Wh-questions</i>	33
3.4 KNOWLEDGE REPRESENTATION METHOD	34
3.4.1 <i>The Choice of Knowledge Representation</i>	34
3.4.2 <i>The Knowledge Base and the Inference Machinery</i>	35
3.5 SENTENCE GENERATION METHOD	36
3.5.1 <i>An Overview of the Sentence Generation</i>	36
3.6 POST-PROCESSING METHOD	37
4 IMPLEMENTATION	39
4.1 THE INPUT ANALYSIS MODULE	39
4.1.1 <i>Reading Input</i>	40
4.1.2 <i>Calling Java and QTAG</i>	40
4.1.3 <i>Polishing QTAG's Output</i>	40
4.1.4 <i>Java and Prolog: a Problematic Friendship</i>	42
4.2 THE KNOWLEDGE REPRESENTATION MODULE	42
4.2.1 <i>Storing Information in the KB</i>	43
4.2.2 <i>Recognising Wh-words</i>	46
4.2.3 <i>Defining Topic</i>	46

4.2.4 <i>Selecting Information</i>	49
4.2.5 <i>A Guided Tour through a KB Search</i>	50
4.3 THE GENERATION MODULE.....	53
4.3.1 <i>An Overview of the Generation Module</i>	53
4.3.2 <i>A Detailed Description of the Whole Generation System</i>	55
4.4 THE POST-PROCESSING MODULE.....	61
4.4.1 <i>Prepositions</i>	63
4.4.2 <i>Indefinite Articles</i>	64
4.4.3 <i>Numerals</i>	64
4.4.4 <i>Anaphors as Subjects</i>	65
4.4.5 <i>Conjoining Sentences</i>	65
4.4.6 <i>Printing Output</i>	66
4.5 RESULTS.....	66
5 DISCUSSION	71
5.1 THE INPUT ANALYSIS MODULE	71
5.2 THE SENTENCE GENERATION MODULE.....	72
5.3 THE LACK OF A DISCOURSE MODULE	74
5.4 MODULARITY	76
5.5 CONTRIBUTIONS AND LIMITATIONS	78
6 CONCLUSION.....	81
7 FURTHER WORK.....	83
8 SUMMARY	85
REFERENCES.....	86
APPENDIX A	
APPENDIX B	
APPENDIX C	
APPENDIX D	

Appendix A

The original dialogue

When playing the game, the menu that comes up when encountering Tobias contains the lines A1, A8 and A9. The other ones are lines that either come up in subdialogues (another menu pops up with new lines) or lines that are given by April automatically, without the player actually choosing them. In this scene, the lines A3 – A7 and A10 – A11 are subdialogues, while the lines A2 and A12 are automatic lines.

- A1: I have to find the plate that unlocks the Guardian's tower.
T1: The plate that is the key? Yes, you will need it. It can even restore the Balance, if the new Guardian follows it to the tower, of course. Do you really want to find the plate on your own?
A2: I have to.
T2: When the earth was divided and the land of the Guardian was created, a plate was made. The plate would serve two purposes, as key to the tower of the balance and as replacement for the plate that already existed in the tower, should that one be destroyed. The tower is now abandoned and locked and the old plate is broken. I think the time has come to find the new plate and use it.
A3: Where is the plate now?
T3: It was first stored out in the open, in the enclave of the watchmen in Marcuria, but when thieves tried to steal it, it was removed and split up into four parts. Each part was given to one of the four enchanted people in Stark – people who didn't have anything to gain from the disturbance of the Balance.
A4: Which people were given a part of the plate?
T4: I'm not sure that anyone remembers that, but I'm sure it is in the scriptures.
A5: What scriptures?
T5: The scriptures of the Balance. There are thirteen of them.
A6: Where can I find these scriptures?
T6: Visit the enclave of the watchmen. Go to the library in the enclave and talk to Minstrum Yerin. Tell him I sent you.
A7: Where can I find these scriptures?¹
T7: Again, you will find the scriptures in the enclave of the watchmen. Talk to Minstrum Yerin.
A8: I also must find the entrance to the land of the Guardian.
T8: There is one, but I don't know where it is. The only entrance I can think of should be where the dichotomy was created, where the tower later was built.
A9: I have to find the two dragons that live in Arcadia.
T9: The Draic breed? They have been here since long before our time. The breed was the greatest contributors to the dichotomy and saved mankind from termination.
A10: How can I find out more about the dragons?
T10: Scriptures! There is one called "The secret of the Draic breed", which is very informative.
A11: Where can I find this scripture?
T11: Again, you will find the scriptures in the enclave of the watchmen. Talk to Minstrum Yerin.
A12: Thank you.
T12: I'm glad to help

Our way of representing the propositional content of the original dialogue, somewhat rewritten to make some of the structures more simple:

¹ This is actually just a repetition of the previous question. We included this to show what happens in the game, if the player chooses the same question more than once.

Topics	Information
plate	<p>The plate was divided in four parts. The parts were given to four enchanted people. The plate can restore the Balance. The plate is the key. The plate opens the tower.</p>
Guardian	<p>The Guardian protects the Balance The Guardian disappeared in Stark. You must find the Guardian.</p>
tower	<p>The tower was built for the Guardian. The Guardian must return to the tower. The tower was risen in the dichotomy's creation.</p>
Balance	<p>The earth was divided in two worlds. Arcadia is the magical world. Stark is the logical world. A balance must exist between the worlds. The Balance is a force. The Balance controls the worlds. The Balance prevents chaos.</p>
watchmen	<p>The watchmen guard the worlds. The watchmen live in both worlds. The watchmen have a library. The library contains much useful information.</p>
library	<p>The library is in the watchmen's enclave. Yerin is the librarian. You must talk to Yerin.</p>
scriptures	<p>The scriptures are in the library. The scriptures will tell about the enchanted people. You must read the scriptures.</p>
entrance	<p>The entrance is in the tower.</p>
dragons	<p>The dragons saved the human race. The dragons contributed to the dichotomy. The scriptures will tell about the dragons.</p>
dichotomy	<p>The earth was divided in two worlds. The scriptures will tell about the dichotomy.</p>
Yerin	<p>Yerin is the librarian. Yerin is a watchman. Yerin is in the watchmen's enclave.</p>
enclave	<p>The enclave is the watchmen's state.</p>

Appendix B

The entire KB.

```
% kb.pl: (c) Lena Carlsson & Jenny Gustafson, Spring 2000
```

```
/*  
This file contains the knowledge of the game character.  
x is a place holder for the topic.  
***/
```

```
kb(arcadia, [(g,[x,magical,world])  
            ]).
```

```
kb(stark, [(g, [x,logical,world])  
          ]).
```

```
kb(plate, [(where,[x,in,arcadia]),  
          (g, [x,divided,four,parts]),  
          (g, [parts,given,enchanted,people]),  
          (g, [x,can,restore,balance]),  
          (g, [x,key]),  
          (g, [x,opens,tower])  
          ] ).
```

```
kb(guardian, [(where, [x,disappeared,stark]),  
             (who, [x,protects, balance]),  
             (g, [tower,built,x]),  
             (g, [x,return,to,tower]),  
             (g, [you,must,find,x])  
             ]).
```

```
kb(tower, [(where, [x,risen,dichotomys,creation]),  
          (g, [x,built,guardian]),  
          (g, [guardian,return,to,x])  
          ]).
```

```
kb(balance, [(g, [x,force]),  
            (g, [x,controls,worlds]),  
            (g, [x,prevents,chaos]),  
            (g, [earth,divided,two,parts]),  
            (g, [x,exist,between,worlds]),  
            (g, [stark,logical,world]),  
            (g, [arcadia,magical,world])  
            ]).
```

```
kb(watchmen, [(where,[x,exist,in,both,worlds]),  
             (g, [x,have,library]),  
             (g, [x,guard,worlds])
```

```
    ]).

kb(people, [(g, [scriptures,tell,about,x])
            ]).

kb(library, [(where, [x,in,watchmens,enclave]),
             (g, [x,contains,much,useful,information])
            ]).

kb(scriptures, [(where, [x,in,library]),
                (g, [x,tell,about,enchanted,people]),
                (g, [you,read,x]),
                (g, [yerin,protects,scriptures])
               ]).

kb(entrance, [(where, [x,in,tower])
              ]).

kb(dichotomy, [(where, [scriptures,tell,about,x]),
               (g, [earth,divided,two,worlds])
              ]).

kb(dragons, [(where, [scriptures,tell,about,x]),
             (g, [x,saved,human,race]),
             (g, [x,contributed,dichotomy])
            ]).

kb(yerin, [(who, [x,librarian]),
           (where, [x,in,library]),
           (g, [x,watchman]),
           (g, [you,talk,to,x])
          ]).

kb(enclave, [
            (g, [x,has,library]),
            (g, [x,watchmens,state])
            ]).
```

Appendix C

The DCG and lexicon used for sentence generation

```
% grammar.pl: (c) Lena Carlsson & Jenny Gustafson, spring 2000

/*****
*****
This file contains the DCG used for generating English
sentences.
*****
*****/

s(s(NPTree,VPTree)) -->np(NPTree,Num,Pers),
                        vp(VPTree,Num,Pers).

np(np(DET,NTree),Num,Pers)-->det(DET,Num), n(NTree,Num,Pers).
np(np(PNTree),Num,Pers)--> pn(PNTree,Num,Pers).
np(np(DET,NTree),Num,Pers)-->det(DET,Num),
                        nbar(NTree,Num,Pers).
np(np(PronTree),Num,Pers)--> perspro(PronTree,Num,Pers).

nbar(nbar(ADJTree,NTree),Num,Pers) --> adj(ADJTree),
                        n(NTree,Num,Pers).
nbar(nbar(GENTree,NTree),Num,Pers) --> ngen(GENTree),
                        n(NTree,Num,Pers).

vp(vp(VTree,PPTree),Num,Pers) --> vbar(VTree,Num,Pers),
                        pp(PPTree).
vp(vp(VTree,NPTree),Num,Pers) --> vbar(VTree,Num,Pers),
                        np(NPTree,_,_).
vp(vp(VTree,ADVPTree),Num,Pers) --> vbar(VTree,Num,Pers),
                        advp(ADVPTree).
vp(vp(TVTree,NPTree),Num,Pers) --> tv(TVTree,Num,Pers),
                        np(NPTree,_,_).
vp(vp(IVTree,PPTree),Num,Pers) --> iv(IVTree,Num,Pers),
                        pp(PPTree).
vp(vp(VTree,PPTree),Num,Pers) --> vpass(VTree,Num,Pers),
                        pp(PPTree).

vbar(vbar(COP),Num,Pers) --> cop(COP,Num,Pers).
vbar(vbar(AUX,VTree),_,_) --> aux(AUX), vinf(VTree).
```

```
vpass(vpass(BETree,PCPTree),Num,Pers) --> be(BETree,Num,Pers),
      vpcp(PCPTree).
```

```
pp(pp(P,NPTree)) --> p(P), np(NPTree,_,_).
```

```
advp(advp(ADV,PPTree)) --> adv(ADV), pp(PPTree).
```

```
adj(adj(Word)) --> [Word], {lex(adj,Word)}.
```

```
ngen(ngen(Word)) --> [Word], {lex(ngen,Word)}.
```

```
det(det(Word),Num) --> [Word], {lex(det,Num,Word)}.
```

```
perspro(perspro(Word),Num,Pers) --> [Word],
      {lex(perspro,Num,Pers,Word)}.
```

```
pn(pn(Word),Num,Pers) --> [Word], {lex(pn,Num,Pers,Word)}.
```

```
n(n(Word),Num,Pers) --> [Word], {lex(n,Num,Pers,Word)}.
```

```
adv(adv(Word)) --> [Word], {lex(adv,Word)}.
```

```
vinf(vinf(Word)) --> [Word], {lex(vinf,Word)}.
```

```
iv(iv(Word),Num,Pers) --> [Word], {lex(iv,Num,Pers,_,Word)}.
```

```
tv(tv(Word),Num,Pers) --> [Word], {lex(tv,Num,Pers,_,Word)}.
```

```
vpcp(vpcp(Word)) --> [Word], {lex(vpcp,Word)}.
```

```
aux(aux(Word)) --> [Word], {lex(aux,Word)}.
```

```
be(be(Word),Num,Pers) --> [Word], {lex(be,Num,Pers,Word)}.
```

```
cop(cop(Word),Num,Pers) --> [Word], {lex(cop,Num,Pers,Word)}.
```

```
p(p(Word)) --> [Word], {lex(p,Word)}.
```

```
% lexicon.pl: (c) Lena Carlsson & Jenny Gustafson, spring 2000
```

```
/*  
*
```

```
This file contains the lexicon, used by the sentence  
generation predicates.
```

```
*****  
/
```

```
%DETERMINERS
```

```
lex(det,_,the).
```

```
lex(det,pl,four).
```

```
lex(det,pl,two).
```

```
lex(det,sg,a).
```

```
lex(det,pl,both).
```

```
lex(det,sg,much).
```

%PERSONAL PRONOUNS
lex(perspro,_,0,you).

%PROPER NOUNS
lex(pn,sg,3,arcadia).
lex(pn,sg,3,stark).
lex(pn,sg,3,yerin).

%NOUNS IN GENITIVE FORM
lex(ngen,dichotomys).
lex(ngen,watchmens).

%ADJECTIVES
lex(adj,encharnted).
lex(adj,useful).
lex(adj,human).
lex(adj,logical).
lex(adj,magical).

%NOUNS
lex(n,sg,3,plate).
lex(n,pl,0,parts).
lex(n,_,0,people).
lex(n,sg,3,key).
lex(n,sg,3,balance).
lex(n,sg,3,tower).
lex(n,sg,3,guardian).
lex(n,sg,3,creation).
lex(n,sg,3,dichotomy).
lex(n,sg,3,earth).
lex(n,pl,0,worlds).
lex(n,sg,3,world).
lex(n,sg,3,magic).
lex(n,sg,3,logic).
lex(n,sg,3,force).
lex(n,sg,3,chaos).
lex(n,sg,3,watchman).
lex(n,pl,0,watchmen).
lex(n,pl,0,guards).
lex(n,sg,3,library).
lex(n,sg,3,information).

lex(n,pl,0,scriptures).
lex(n,sg,3,enclave).
lex(n,sg,3,city).
lex(n,sg,3,keeper).
lex(n,sg,3,entrance).
lex(n,pl,0,dragons).
lex(n,sg,3,race).
lex(n,sg,3,state).
lex(n,sg,3,librarian).
lex(n,sg,3,east).

%ADVERBS

lex(adv,east).

%INFINITE VERBS in infinitive

lex(vinf,tell).
lex(vinf,talk).
lex(vinf,return).
lex(vinf,restore).
lex(vinf,find).
lex(vinf,be).
lex(vinf,exist).
lex(vinf,read).

%FINITE INTRANSITIVE VERBS

lex(iv,_,_,imp,disappeared).
lex(iv,_,0,pres,live).
lex(iv,_,_,imp,contributed).

%FINITE TRANSITIVE VERBS

lex(tv,sg,3,pres,opens).
lex(tv,sg,3,pres,protects).
lex(tv,sg,3,pres,controls).
lex(tv,sg,3,pres,prevents).
lex(tv,_,0,pres,have).
lex(tv,_,_,imp,saved).
lex(tv,sg,3,pres,contains).
lex(tv,_,0,pres,guard).
lex(tv,sg,3,pres,has).

%VERBS IN PARTICIPLE - for passive voice

lex(vpcp,divided).
lex(vpcp,built).
lex(vpcp,risen).
lex(vpcp,given).

%AUXILIARIES

lex(aux,can).

lex(aux,must).

lex(aux,will).

%PASSIVE BE-FORMS

lex(be,sg,3,was).

lex(be,_,0,were).

%COPULA

lex(cop,sg,3,is).

lex(cop,_,0,are).

%PREPOSITIONS

lex(p,at).

lex(p,in).

lex(p,to).

lex(p,for).

lex(p,between).

lex(p,about).

lex(p,from).

lex(p,of).

APPENDIX D

This Appendix includes the **main** predicates in the Generation Module.

```
% Generates all possible sentences from the input words

generate_all(Input,CorrectSentences):-
    findall(Sentence,generate(Input,Sentence),ListOfSentences),
    parse(ListOfSentences,CorrectSentences), !.

% Generates one sentence from a list of input words

generate(InputList,Sentence):-
    lookup(InputList,POSList),
    remove_features(POSList,StrippedPOSList,_),
    abstract_and_expand(StrippedPOSList,ExpandedList,MissingCat
),
    if(corr(StrippedPOSList,ExpandedList,CorrelatedList),
        nth(Position,CorrelatedList,MissingCat),
        nth(Position,ExpandedList,MissingCat)),
    findall((Word,Num),(apply(MissingCat,_ ,Num,Word,_)),List),
    insert_to_inputlist(List,InputList,Position,NewInputList),
    if((abstract_and_expand(ExpandedList,[s],_)),
        ((Sentence=NewInputList)),
        (generate(NewInputList,Sentence))).

% First checks for phrases that can be made without additional
% words and then expands with possible added words and POSs.
% For example abstract_and_expand([n,p,pn],OutputList,Missing)
% → OutputList = [det,n,pp], Missing = det

abstract_and_expand(Xs,Zs,Missing):-
    check_all(Xs,Ys),
    insert_all_links(Ys,Zs,Missing).

% Takes a POSlist and tries to make phrases directly from
% these POSs. For example check_all([n,p,pn], X) → X =
% [n,p,np]

check_all([X],[X]).
check_all(Xs,Ys):- no_more_phrases(Xs,Ys),!.
check_all(Xs,CL):-
    transform(Xs,Ys),
```

```

    check_next(Ys,CheckedList),
    check_all(CheckedList,CL),!.

% Checks if the two last POSs in a list can be turned into a
% phrase.

check_next(Xs,Xs):- out_of_phrases(Xs),!.
check_next([X],[X]).
check_next(Xs,Phrase):-
    delete_last(Xs,NewXs,LastElement),
    delete_last(NewXs,ReducedList,SecondLastElement),
    if((fras(P,SecondLastElement,LastElement)),
        (removelist(ReducedList,NewList),glue(NewList,P,Phrase)),
        check_next(NewXs,TempPhrase),
        glue(TempPhrase,LastElement,Phrase))).

% Finds the additional POSs that are needed to form sentences.

insert_all_links([s],[s],[]).
insert_all_links(Xs,Ys,Missing):-
delete_last(Xs,NewXs,LastElement),
    delete_last(NewXs,ReducedXs,SecondLastElement),
    link(SecondLastElement,LastElement,Missing,_,List),
    append(ReducdXs,List,Ys),!.

% Inserts the word if the position is adequate and returns a
% new list with the word inserted.

insert_to_inputlist(Xs,Ys,Position,NewList):-
    compare_args(Xs,_,Agreed),
    insert_word(Agreed,Position,Ys,NewList).

% Takes a POSlist and checks if any more phrases can be made
% from the POSs.

no_more_phrases(Xs,Ys):-
    if((transform(Xs,Ys),Xs\==Ys),(out_of_phrases(Ys),Ys=Xs),
        (Ys=Xs,out_of_phrases(Ys))).

```

```

% A boolean predicate which succeeds if it is impossible to
% make any further abstractions in the POSlist.

out_of_phrases([]).
out_of_phrases([X]).
out_of_phrases([X,Y|Xs]):-
    \+(\^(P,fras(P,X,Y)),out_of_phrases([Y|Xs])).

% Transforms POS that on their own can make up other phrases,
% by abstraction.

transform([],[]).
transform([X|Xs],[Y|Ys]):- fras(Y,X,[]),
    transform(Xs,Ys),!.
transform([X|Xs],[X|Ys]):- transform(Xs,Ys).

% Deletes the last POS of a POSlist and returns the remainder
% of the POSlist and the deleted POS.

delete_last([],[],_).
delete_last([X|Xs],Zs,Y):-
    reverse([X|Xs],[Y|Ys]),
    reverse(Ys,Zs).

% Takes a list of input words and returns a list with the
% corresponding POSs.

lookup([],[]).
lookup([X|Xs],[POS,Num,Pers|POSS]):- lex(POS,Num,Pers,X),
    lookup(Xs,POSS).
lookup([X|Xs],[POS,_,_|POSS]):- lex(POS,X),
    lookup(Xs,POSS).
lookup([X|Xs],[POS,Num,Pers|POSS]):- lex(POS,Num,Pers,_,X),
    lookup(Xs,POSS).
lookup([X|Xs],[POS,Num,_|POSS]):- lex(POS,Num,X),
    lookup(Xs,POSS).

% Correlates the POSlist (Xs) and the expanded list (Ys) and
% returns a correlation (List).

corr(Xs,Ys,List):-
    correlate(Xs,Ys,Zs),
    flatten(Zs,List).

% Finds possible correlations and returns a correlated list,

```

```

% such as in
% correlate([det,n,det,n],[np,vbar,np],[[det,n],vbar,det,n]).

correlate(Xs,[Y],Match):- match(Y,Xs,Match,[ ]).
correlate(Xs,[Y|Ys],[Match|Zs]):-
    match(Y,Xs,Match,Rest),
    if(\+correlate(Rest,Ys,Zs),
        (head(Ys,Y2),
         append([Y2],Rest,Zs)),
        correlate(Rest,Ys,Zs)).

head([X|_],X).

% Flattens a list, in the way that it removes embedded lists
% and returns only one flat list, such as in
% flatten([[det,n],vbar,det,n],[det,n,vbar,det,n]).

flatten([],[]):-!.
flatten(X,[X]):-
    atomic(X).
flatten([Left|Right], Flat) :-
    flatten(Left, LeftFlat),
    flatten(Right, RightFlat),
    append(LeftFlat, RightFlat, Flat).

```