Project Report

45073 Computer Science, Projects

Natural Language Interfaces to Rule Based Systems

Anders Blehr
Knowledge Systems Group
Division of Computer Science & Telematics
The Norwegian Institute of Technology
The University of Trondheim

May 4, 1992

Abstract

This paper constitutes my final report in course *45073 Computer Science, Projects* at The Norwegian Institute of Technology, Division of Computer Science & Telematics. The aim of my work has been to develop and implement a prototype of a natural language based expert system shell. In connection with this, I have sought to address some of the general problems related to the field of natural language processing, as well as to define a suitable platform on which to base my implementation.

Contents

Chapter 1

Introduction

1.1 Motivation

The Understanding Computer (*TUC*) is a project initiated by Tore Amble at the Knowledge Systems Group of The Division of Computer Science & Telematics (IDT) at The Norwegian Institute of Technology (NTH), Trondheim, Norway.

TUC is a successor of the Nordic cooperation project *HSQL*, which was a prototype for answering queries to a relational database, given in any of the Scandinavian languages.

The goal of the project is to investigate *"how a naturally readable language can be a unifying framework for knowledge representation and natural language interfaces,* [that is, whether] *natural language is able to express knowledge in a way that, apart from its understandability, also is sufficient for an automated reasoning system to draw the right conclusions"* ([1]).

In connection with *TUC*, Tore Amble has developed a logical language *SOLON* (*Second Order Logic for Natural Language*), which allows flexibility with respect to the number and types of arguments and which eliminates as many variables as possible from expressions. For a further description of *TUC* and its foundations, see [1].

Initially, it was the intention that I should work directly with *TUC*, but given its complexity and the limited time I had available, it was decided that I should develop an independent system, and that Tore Amble, who has implemented the current version of *TUC*, and who doubtlessly has the most in-depth knowledge of it, incorporate those parts of my work he sees fit into it.

1.2 Structure of the Paper

In Chapter 2, an introduction to natural languages and natural language systems is given, as well as a description of rule based systems and brief introduction to predicate logic. In Chapter 3, my system, *NaSh*, a natural language based expert system shell, is described. In Chapter 4 the conclusions I have drawn while working with the project are presented.

1.3 The Project Environment

All computational work was done using the *SPARC* and *HP* workstations at IDT.

NaSh was implemented solely in *SICStus Prolog*, a *WAM*-based implementation of Prolog made at The Swedish Institute of Computer Science, Kista, Sweden (see [4]).

This document was prepared with Leslie Lamport's LaTeX document preparation system (see [9]) and Donald Knuth's TeX typesetting system on which LaTeX is based.

Lastly, I would like to thank my supervisor Tore Amble for his enthusiasm and inspiring guidance.

Chapter 2

Background

In this chapter, a brief introduction is given to some of the issues pertaining to the problems of natural language processing, intelligent interfaces, rule based systems and predicate logic.

2.1 Natural Language Systems

2.1.1 What Constitutes a Natural Language?

Natural languages, as opposed to artificial languages (e.g. programming languages), are the languages human beings use in their daily life to communicate all sorts of information about the world. Thus Livian, Sorb, Basque, Norwegian, English and all other known languages (as well as those not yet discovered) constitute natural languages.

What characterizes natural languages is that they are never stable. They are the result of a continued evolution, adapting to the ever changing set of demands characterized by the cultural as well as social setting in which each language is used, adopting words and constructs from other languages, but also evolving in what may seem to be a completely random manner. This evolution has by no means come to an end. On the contrary, the large amount of information available in today's society combined with the extreme ease of communication has caused an acceleration of this process that is unparalleled in history. As a result of this evolution, natural languages have come to be extremely powerful, both with respect to expressivity and flexibility. On the other hand, the nondeterminism and inherent ambiguity of natural languages, although being useful to humans in their daily use of the language, cause serious problems for automated natural language systems.

Nondeterminism and Ambiguity

An inherent feature of natural languages is that it is not possible *a priori* to determine the set of well-formed sentences of which a language consists. This can be summarized as follows (from [6]):

- The vocabulary of a natural language is not completely known, in particular because of the existence of specialized vocabularies: technical, medical, local, etc.
- The set of constructions is itself not completely predetermined.
- The set of senses attributed to each word is also not completely predetermined, especially because a word often does not really have a precise sense outside a particular context.

In addition, identical sentences can contain different meanings depending on the context in which they occur. Take for instance the sentence

He saw her shaking hands.

Whether he (whoever *he* may be; this is an example of what is called *anaphoric references*, i.e., the meaning of a sentence depends on knowledge gained from previous sentences) saw her shaking hands with someone, or saw that her hands were shaking, is not evident from the sentence. (In speech this ambiguity is usually avoided due to the fact that human beings use stress and intonation as a means of conveying the intended meaning to the receiver(s) of their utterances. This is an aspect of the act of communicating which is completely lost in written language).

Another phenomenon is that to extract the correct meaning from sentences, it is not always sufficient only to have knowledge about the meanings of each word. Consider the sentences

- *John saw the boy in the park with a telescope.*
- *John saw the boy in the park with a dog.*
- *John saw the boy in the park with a statue.*

In these three sentences, only the last word of the last prepositional phrase differs, and from a merely syntactic viewpoint it is not possible to deduce whether in each case this prepositional phrase is to be attributed to John's seeing act, the boy or the park. In fact, in each case, all attributions can be made to make sense if one carefully defines the contexts in which they appear. On the other hand, it is not very probable that John used a dog to see the boy in the park, or that the boy was carrying a statue with him through the park. Still it is important, although one knows that it is most likely that John used a telescope to see the boy, not to exclude the possibility that he in fact *did* use the dog (i.e., he was blind and his guide dog was trained to give him a certain signal when it saw a boy in the park).

In other words, to extract the intended meaning from a sentence, it is necessary for the receiver, in addition to having semantic knowledge about each word, also to have knowledge about the world in which the language is used, and to be able to (unconsciously) assign probability measures to each possible interpretation of the sentence in order to arrive at a decision as to which one is the right one.

2.1.2 Natural Language Processing

Natural Language Processing (NLP) is the field of which the objective is to make systems that are able to understand and reason on the basis of natural languages. It is useful to divide the entire NLP problem into two tasks, of which the first is a subset of the second (from [15]):

- Processing written text, using lexical, syntactic and semantic knowledge of the language as well as the required real world information.
- Processing spoken language, using all the information needed above, plus additional knowledge about phonology as well as enough added information to handle the further ambiguities that occur in speech.

The field of speech recognition is mainly concerned with being able to deduce from audio signals the sequence of words which make up an utterance. Since this sequence is readily given in written text, it is only natural that by far the most research has been done in the field of processing written language, of which NLP more or less has become a synonym.

The principal areas of research in NLP are (from [6]):

- Developing and modeling linguistic systems.
- Conceiving and implementing models and systems of NLP.
- Evaluating such systems from the point of view of human-machine interfaces.

The necessity of developing new linguistic systems is due to the fact that existing linguistic systems have usually been developed without the needs of NLP in mind. That is, to be able to develop a working model of a linguistic system, it is necessary that the linguistic system be suited for modeling.

The Process of Understanding Natural Language

The term *understand* in this context is to be understood as the ability of an NLP-system to respond to statements and queries given to it in natural language as if these responses were the result of a human-like reasoning process. The extent to which an NLP-system succeeds in seeming to understand and act according to its input is closely connected with its ability to act in correspondence with the principles of *cooperativity*, described briefly in Section 2.1.3, and thoroughly treated in [11].

The process of understanding natural language (written text) can be divided into the following individual, and more or less independent, steps:

Morphological Analysis Individual words are decomposed into their components, inflected words into their stems, and derived words are traced back to their sources.

Syntactic Analysis The string of words is analyzed to see whether they represent correct sentences in the given language. If they do, they are transformed into structures showing how the individual words of each sentence relate to each other.

Semantic Analysis The structures created during the syntactic analysis are analyzed to extract from them the meaning of the sentences. Two important issues must be addressed (from [15]):

- Map individual words into appropriate objects in the knowledge base or database. If no such mapping is found, the sentence may or may not be refused, depending on the aims of the particular application.
- Create the correct substructures to correspond to the way the meanings of the individual words combine with each other.

Pragmatic Analysis The structure representing what was said is analyzed to determine which action the system should take. I.e., a statement like *"It is cold in here"* could be interpreted as a simple declarative statement to the fact that it indeed *is* cold, but it could also be (and should be, according to the cooperativity principles) interpreted as a request to shut the window, given that there is an open window in the room where the statement is uttered.

The sequence in which these steps are performed is not necessarily the same from one system to another, nor are the boundaries between them. Also, if it is found that a given system would work according to its specification without going through all the steps when analyzing its input, it is no requirement that it do. Still, every full-fledged NLP-system has to have its input thoroughly analyzed to be certain to behave correctly in all situations, and is thus subject to carrying its input through all the steps.

2.1.3 Cooperative Responses

In human to human conversation, a lot of knowledge is passed on without being explicitly stated. In spoken conversation, the amount of information conveyed by stress, intonation, facial expressions and body language in general often by far exceeds the amount contained in the few words actually spoken. Since we are only concerned here with written language, all this information is lost to us, but even what appears to be a short declarative statement in written language, can convey a large body of hidden and implicit information, information which has to be inferred in order for the system to be able to act cooperatively and to the user's satisfaction. From [6]:

> [Human conversational exchanges] *are characteristically cooperative efforts and each participant recognizes in them a common purpose or a set of purposes or, at least, a mutually accepted direction. Unless exceptional circumstances prevail, each participant in a conversation is responsible for detecting and correcting misconceptions that might otherwise occur, and expects the other participants to do the same.*

These implicit behaviour rules have been formalized by Grice, who in [7] suggests several maxims constituting what he calls the *Cooperation Principle*. The maxims of quality, quantity, relation and manner are the ones of concern here:

The maxim of quality This maxim requires that one try to make a truthful contribution:

- Do not say what you believe to be false.
- Do not say that for which you lack adequate evidence.

The maxim of quantity This maxim relates to the amount of information contained in a contribution:

- Make your contribution as informative as is required.
- Do not make your contribution more informative than is required.

The maxim of relation This maxim requires that a contribution be relevant to the conversation.

The maxim of manner This maxim relates to questions of presentation:

- Avoid obscurity of expression.
- Avoid ambiguity.
- Be brief.

The other maxims suggested by Grice address aesthetic, social and moral questions, but in general the four maxims above are sufficient to guide a meaningful discourse. The Cooperation Principle was intended merely as a description of how humans behave in a conversation, but it has come to be widely accepted also as a model on which to base intelligent interfaces to NLP-systems.

By *cooperative responses* we mean responses that correspond as closely as possible to those we would expect to get from a person trying to understand and reply to the queries of another person. In connection with NLP-systems, the aim is, in other words, to make the system behave as much in accordance with the Cooperation Principle as possible.

2.2 Rule Based Systems

When we start out on the task of solving a particular problem, it is important that we know what the initial situation is (that is, the situation defining the problem), as well as what final situations constitute acceptable solutions to the problem. Along the way from the initial situation to an acceptable final situation, we pass through a set of intermediate steps, each leading from one situation to another. It is convenient that all of these intermediate situations, as well as the initial and final situation(s), be classified as *states*, and that the process of moving from the initial to a final state be viewed as moving through a *state space*. Further, we need a set of *rules* describing the legal moves, given the current state. In any state, choosing one rule over another may bring us closer to or farther away from a solution, depending on the specific rule. In other words we need a control strategy to help us choose an appropriate rule. When no direct method for choosing a rule is known, as is usually the case, *search* has to be applied. Search also provides a framework into which more direct methods for solving subparts of a problem can be embedded. Thus the process of finding a solution to a given problem can be classified as searching through a state space.

2.2.1 Production Systems

Production systems provide a means for describing and performing the search process described above. A production system consists of (from [15]):

- A set of rules, each consisting of a left side (a pattern) that determines the applicability of the rule and a right side that describes the operation to be performed if the rule is applied.
- One or more knowledge/databases that contain whatever information is appropriate for the particular task.

- A control strategy that specifies the order in which the rules will be compared to the database and a way of resolving the conflicts that arise when several rules match at once.
- A rule applier.

This definition is very general, and it encompasses a great number of systems, including a family of complex, often hybrid, systems known as *expert systems* (see Section 2.2.2).

Control Strategies

As mentioned above, we need a control strategy to help us decide which rule to apply, given the current state of a search for a solution of a problem. How these decisions are made has a great impact on how quickly, if at all, a solution is reached. Two requirements the control strategy has to fulfill, are that it cause *motion* and that it be *systematic*. The first requirement ensures that a solution sooner or later will be arrived at, given that it exists, and the second requirement ensures that a certain path in the search space not be explored more than once.

Several search strategies have been suggested, the most well-known being

Depth-first search Construct a tree with the initial state as its root. Generate a child by applying one of the applicable rules to the initial state. Pursue a single branch of the tree by repeating this process at the lastly generated leaf node until a goal is produced or it is decided to terminate the branch. If the branch is terminated, backtrack to the first predecessor which has not had all of its offspring examined and continue the search from there.

Breadth-first search Generate all the offspring of the root by applying each of the applicable rules to the initial state. Repeat this process for all the leaf nodes until some rule produces a goal.

Heuristic search techniques Sometimes it is required that one compromise the requirements of mobility and systematicity in order to efficiently arrive at a solution which no longer is guaranteed to be the best one, but which almost always is a very good one. A *heuristic* is a search technique which accomplishes this, employing what is called a *heuristic function* to decide which rule to choose. The value returned by the heuristic function is a measure of the desirability of exploring the given path. "*The purpose of a heuristic function is to guide the search process in the most profitable direction by suggesting which path to follow first when more than one are available*" (from [15]).

Most complex problems, and that means most problems of practical interest, need to employ heuristic search techniques in order to be solved efficiently, and the process of problem solving can thus more accurately be described as a heuristic search process rather than merely a search process.

2.2.2 Expert Systems

As mentioned in Section 2.2.1, expert systems constitute a subset of the family of production systems. An *expert system* can be labeled as a system which guides the user in solving problems which normally requires the intervention of a human expert in the field.

To solve an expert-level problem, an expert system needs to employ a powerful reasoning system (*inference engine;* [2]) as well as to have access to a substantial domain-specific *knowledge base*. To be convincing as to its conclusions, it is also important that the user of the expert system be able to interact with it easily, and that he, when it has arrived at a conclusion, can be given an explanation of how the conclusion was arrived at, that is, that the system be able to explain its reasoning. Moreover, the system has to be able to acquire new knowledge as well as modifications of old knowledge.

Expert System Shells

Expert systems are employed in a wide variety of domains, including medical diagnosis, legislation and decision making, just to mention some. But even though the domains of different expert systems have little or nothing in common, they are generally based on the same principles. Thus it is common to separate the domain specific parts (typically the knowledge base) of an expert system from the non-domain specific "core", which usually is implemented separately as a general purpose expert system *shell*. The tasks performed by the expert system shell are typically (see [2]) those of the inference engine, search processing and dialogue handling (user interface).

From an NLP-point of view, the ideal expert system shell would be one in which all interaction with the user as well as knowledge acquisition and explanation are done in natural language, and which still contains all the features of present day shells. Whether this is possible, is the question which the *TUC*-project seeks to find some of the answers to.

2.3 Logic Foundations

For a formal definition of the terms used in this section, see for instance [8] or [12].

2.3.1 Predicate Logic

First order predicate logic, or simply *predicate logic*, is a branch of the general science of logic, and offers a powerful way of deriving new knowledge from old by means of what is called *mathematical deduction*.

The counterpart in predicate logic of sentences in natural language is *well-formed formulas*, or simply *formulas*. Well-formed formulas are built from *constants*, *variables*, *functors*, *predicate symbols* and recursively from other well-formed formulas. Variables can be either *free* or *bound*. If they are bound, they are either *existentially* or *universally quantified*. The symbols denoting existential and universal quantification are $\exists$ and $\forall$, respectively. Other symbols used in predicate logic, are the logical *connectives*: *conjunction* ($\wedge$), *disjunction* ($\vee$), *negation* ($\neg$) and *implication* ($\rightarrow$).

Given the above, the natural language sentence "Socrates is a man" can be represented by the well-formed formula

$$man(socrates).$$

Similarly, the following well-formed formulas represent, from top to bottom, the sentences "all men are mortal", "some pigs have wings", "every man loves a woman" and "not all birds can fly":

$$\begin{gathered}(\forall x)(man(x) \rightarrow mortal(x)) \\ (\exists x)(pig(x) \wedge have(x, wings)) \\ (\forall x)(man(x) \rightarrow (\exists y)(woman(y) \wedge love(x,y))) \\ (\exists x)(bird(x) \wedge \neg fly(x)) \text{ or } \neg(\forall x)(bird(x) \rightarrow fly(x)).\end{gathered}$$

The two representations given for the last sentence are of course equivalent. A common, but not universal, pattern of predicate logic ([8]) is recognized here, that the universal quantifier very often is followed by an implication and that the existential quantifier very often is followed by a conjunction. This last property has been utilized in *NaSh* (see Section 3.2.2).

The Internal Knowledge Representation Scheme of *NaSh*

The internal knowledge representation scheme of *NaSh* (*NIKS*) is based on a scheme presented in [14]. The syntax of *NIKS* is very similar to that of predicate logic. In addition to the universal and existential quantifiers, the connectives conjunction, implication and are employed, with a, for obvious reasons, slightly different notation: `forall(X)`, `exists(X)`, `&`, `=>` and `not` for $(\forall x)$, $(\exists x)$, $\wedge$, $\rightarrow$ and $\neg$, respectively. In addition, *NIKS* defines a non-associative binary infix operator `:` for associating quantifiers with the logical expressions they quantify. Thus the examples given in above will look as follows in *NIKS*:

```
forall(X):(man(X)=>mortal(X))
exists(X):(pig(X)&have(X,wings))
forall(X):(man(X)=>exists(Y):(woman(Y)&love(X,Y)))
exists(X):(bird(X)&not fly(X)) or not forall(X):(bird(X)=>fly(X))
```

All the connectives employed in *NIKS* are defined as operators. In order of ascending precedence, they are defined as follows: `=>` is non-associative binary infix, `&` is right-associative binary infix and `not` is prefix. The operator of highest precedence is `:`.

2.3.2 Resolution

The variant of mathematical deduction commonly employed in connection with predicate logic is called *resolution*, and is based on an inference rule called the *resolution principle*. Resolution produces proofs by *refutation*, that is, to prove that a statement is valid an attempt is made to show that the negation of the statement produces a contradiction with the known statements. For a thorough description of how the resolution procedure works, see [12] and [15]. Here we just mention that it requires that all statements be in *clause form*, that is, they should contain no quantifiers and only one connective, the disjunction $\vee$. An algorithm for transforming statements into clause form is given in [15].

To give an informal presentation of how resolution works, imagine that we want to prove that Socrates is mortal given that we know that he is a man and that all men are mortal. I.e., our knowledge base contains the two clauses (which are also given above, although not in clause form)

$$\begin{gathered} man(socrates) \\ \neg man(x) \vee mortal(x), \end{gathered}$$

and we want to prove the clause

$$mortal(socrates).$$

To accomplish this, we try to refute the clause (or rather, the *goal*) $\neg mortal(socrates)$ (i.e., prove that it causes a contradiction with what we already know). When comparing the goal with the knowledge base, we see that it can be resolved with the literal $mortal(x)$, unifying x with *socrates* and leaving us with the contradiction

$$\begin{gathered} man(socrates) \\ \neg man(socrates). \end{gathered}$$

In the next iteration of the the resolution procedure, the empty clause ($\square$) will be produced, which means that we have succeeded in refuting the goal and thus in proving that Socrates indeed *is* mortal.

Chapter 3

A Natural Language Based Expert System Shell

3.1 Aims and Goals

As mentioned in Section 1.1, and further elaborated upon in [1], *TUC* aims at investigating

> *how a naturally readable language can be a unifying framework for knowledge representation and natural language interfaces,* [that is, whether] *natural language is able to express knowledge in a way that, apart from its understandability, also is sufficient for an automated reasoning system to draw the right conclusions.*

My task in this project has been to define a semi-natural language (*SNaL - Semi-Natural Language*) and on the basis of this develop and implement a prototype of a natural language based expert system shell (*NaSh - Natural Language Shell*).

It has been my intention neither to make *SNaL* nor *NaSh* out to be anything more than prototypes. Whether they contain features of interest to Tore Amble in his work with *TUC* - as I hope and believe they do - remains to be seen.

3.1.1 *SNaL* - The Language

The term *SNaL* will be used throughout the rest of this paper to interchangeably refer to the *language* and to the *analyzer* which constructs semantic representations from sentences of the language. In working with *SNaL*, I aimed at making it

- appear, from the user's point of view, as close to standard English as practically tractable. That is, it should never require that the user employ constructs which are illegal in English.
- sufficiently formal. That is, it should be able to represent and convey knowledge and information in a manner which is unambiguous and suited for treatment by an automated reasoning system. In order to avoid the ambiguities of standard English, its vocabulary and set of legal constructs should be finite and well defined.

It is important to keep in mind that, no matter how "natural" *SNaL* may seem at first sight, it still is a *formal* language. No effort has been put into making it more than a very restricted subset of English.

3.1.2 *NaSh* - The Expert System Shell

The aim of *NaSh* has been to make a working prototype of a system containing the basic features of an expert system shell, and of which the dialogue handler is entirely based on *SNaL*. That is, all input to the system (defining rules, supplying knowledge, querying, etc.) as well as all output from the system (answers to queries, explanations, queries to the user, etc) should be represented in *SNaL*.

To support the reasoning process, an internal knowledge representation scheme had to be chosen. I had the option of using Tore Amble's *SOLON* language, which indeed is a powerful tool, but considering that it (or rather its specification) is still under development and thus changing from day to day, I chose to use first order predicate calculus as a platform on which to build the scheme.

3.2 The Language

In this section the semi-natural language employed in *NaSh*, *SNaL*, is described. As mentioned in the previous section, the aim with *SNaL* has been that it be as close to standard English as possible, and that it be sufficiently formal. The emphasis has been on accepting legal constructs rather than rejecting illegal ones.

3.2.1 Linguistic Background

Most of the linguistic material in this section is gathered from [6].

The basic element of any (natural) language is the *word*, which may have associated with it different *inflections* (number, tense, degree, etc.), depending on its function in the language. A *sentence* in a language is a sequence of words, and the particular language defines, by means of abstract rule systems, which sequences constitute legal sentences. Within each sentence, words make up syntactic *constituents*, which in turn can be part of more complex syntactic constituents. The constituents are determined by how strongly the individual words of the sentence are linked to each other (e.g., an adjective will have stronger links with the noun it qualifies than with a preposition). These links are made explicit by the internal structure of the sentence. Simple sentences can be combined or their internal structures can be rearranged in order to form more complex constructions. Basic constituents are built from (centered on) the basic syntactic *categories* (i.e., word classes: nouns, verbs, adjectives, adverbs, prepositions, pronouns, etc.), and are defined in terms of them. Thus we have *noun* phrases, *verb* phrases, *adjective* phrases, *adverbial* phrases, *prepositional* phrases, and so on (the abbreviations *NP*, *VP*, *AdjP*, *AdvP*, *PP* and the like are commonly used). The structure of the constituents is described using *phrase-structure rules*.

As an example, consider English, where a simple sentence can be built from a noun phrase and a verb phrase (using BNF-notation for the phrase-structure rules):

$$S \quad \rightarrow \quad NP,\ VP.$$

Similarly, a noun phrase can be made up from a determiner, an adjective phrase and a noun, a verb phrase from a (transitive) verb and a noun phrase (representing the direct object), and an adjective phrase from either an adjective or the empty string (denoted ϵ):

$$\begin{array}{lll} NP & \rightarrow & Det,\ AdjP,\ N \\ VP & \rightarrow & V,\ NP \\ AdjP & \rightarrow & Adj \\ & | & \epsilon \end{array}$$

Using these simple rules, we can construct sentences like "The old man walked the dog" and "Every man loves a woman".

The incorporation of words is achieved by *lexical insertion rules*, e.g.,

$$N \quad \rightarrow \quad [Noun],$$

where *Noun* is any word in the *dictionary* classified as a noun. The dictionary (or *lexicon*) contains all the words of the given language, along with categorical information and morphological characteristics.

A sentence is said to be *well-formed* if there exists at least one set or sequence of rules which define a complete description of the sentence. If there exist more than one such set or sequence, the sentence is said to be *structural ambiguous* (this corresponds to the inherent ambiguity of any natural language, see for instance the examples in Section 2.1.1).

3.2.2 Definition

For a BNF-description of *SNaL*, see Appendix A.

The largest unit of *SNaL* is the *sentence*. There are two types of sentences: *statements*, representing facts (knowledge) and rules, and *queries*, representing questions, either to the system or to the user. The only form of punctuation marks allowed in *SNaL*, are ".", used to end statements, and "?", used to end queries. For clarity, other punctuation marks may occur in the examples, which all constitute legal sentences in *SNaL*, given that the words they contain are found in the dictionary.

Statements

The statements of *SNaL* can be divided into two distinct categories, *declarative statements*, whereof the set of *implicit rules* constitute a subset, and *formal rules*.

Declarative statements convey some sort of information about the world, and may be conjugated with other declarative statements when different aspects of information about a particular world object (or several world objects) need to be contained in one sentence. The following sentences constitute declarative statements:

John is an old man.
Peter lives with Mary in London.
The woman that Robert loves, loves Frank.
Maurice builds a large house on an open field outside Marseilles.
John looks at Mary and Mary talks to Peter and Peter admires Janice.

A subset of the set of declarative statements is the set of *implicit rules*, which can be used to convey knowledge about specific classes of world objects. An implicit rule is defined by the determiner "every". The justification for this supposition is found in predicate calculus (see Section 2.3.1 and [8]), where it has been pointed out that it is a common, albeit not universal pattern of predicate logic that the universal quantifier be followed by an implication. Examples of implicit rules are:

Every man loves a woman.
Rosette likes every picture that is painted by Cezanne.

Formal rules are constructs of the form *"If <premise> then <conclusion>"*, and are used for defining more complex relations between specific world objects, and particularly for defining general rules to be used for reasoning within a certain domain. Both the *premise* and the *conclusion* have to be declarative statements. The sets of implicit and formal rules are not disjunct, that is, it is often possible to express knowledge about a specific relation either as an implicit or formal rule. It is recommended, though, for the sake of determinism (complex implicit rules may be transformed into logical expressions containing more than one implication, which in *NaSh* represent an unhandled case), that complex relations be expressed by means of formal rules. The following constitute examples of formal rules:

If John is married to Mary, then Mary is married to John.
If Maurice loves Michelle and Michelle is married to Paul, then Maurice is unhappy.

Queries

Queries are used for asking questions to be answered either by the system or the user. What makes queries different from other sentences, (statements, that is), is that they have a different syntactic structure, and that they demand some sort of action to be taken. Queries may belong to one of the following categories:

Aux-prefixed The first word of the query is either "*is*" or "*does*" (plurals are not handled, see Section 3.2.4). Queries belonging to this category may be classified as *confirmative*, in that the answers they demand be either negative ("*no*"-answers) or affirmative ("*yes*"-answers).

Wh-prefixed The first word of the query is a so-called *wh-pronoun* ("*what*", "*who*", "*where*", "*when*", etc.). Queries belonging to this category correspond to the user (or the system) during the reasoning process having come across an unbound variable which is required to be bound in order that a solution be found. The response given is required to cause a binding of this variable, usually to a term supplied as part of the response.

In addition, aux-prefixed queries (to the system) may be prefixed with the keyword "*how*" in order that a proof be supplied (see Section 3.3.3). Examples of queries are (the corresponding declarative statements are given above):

Is John an old man?
Does Peter live with Mary in London?
Who does the woman that Robert loves, love?
What does Maurice build on an open field outside Marseilles?

Using Variables

To attribute knowledge to a specific class of world objects as a whole (e.g., persons, birds, houses, etc.), it would be rather cumbersome explicitly to attribute this knowledge to every known individual belonging to the class. Rather, it would be desirable that it be possible to refer to the class as a whole. Specifically, we need to be able to assert *general rules*, applying to entire classes of world objects, not only to specific individuals of the class. To achieve this, specific syntactic constructs (defined by Tore Amble, and also employed in *TUC*) have been introduced to represent uninstantiated individuals of a class.

Look at the two formal rules listed in the example on page 11. They apply only to John and Mary, and Maurice, Michelle and Paul, respectively, even though they define universal relations, namely that marriage is a commutative relation, and that a man who loves a married woman (who is not his wife) is unhappy (whether this last relation is always true in the *real* world is not of concern here). If we restate these rules as

If any man is married to any woman then this woman is married to this man.

and

If any man loves any woman and this woman is married to any other man,
then this man is unhappy.,

they, if they are universally quantified, still apply to John and Mary and the rest, given that it is known that John is married to Mary, and that Maurice loves Michelle who is married to Paul. In addition, they apply to every other set of individuals fulfilling the same premises. This can be formalized as follows:

The *key structures* "*any*", "*any other*" and "*any third*" define the immediately following noun to quantify a variable. Thus a specific noun may quantify up to three different variables (by defining more key structures, it would be possible that it quantify accordingly many more variables). If a set of variables has been defined by these constructs, they may be referenced at a later time (within the same rule) by applying the key structures "*this*", "*this other*" and "*this third*", together with the quantifying noun. Thus the variables defined by the constructs "*any man*", "*any other piece*" and "*any third river*" may be referenced using the constructs "*this man*", "*this other piece*" and "*this third river*".

It is evident that restricting the words "*any*" and "*this*" to this use only, imposes restrictions on the language we are defining. On the other hand, the advantages of employing a mechanism like this one by far exceed the disadvantages.

3.2.3 Analysis and Generation

SNaL is treated at a purely syntactic level. That is, the semantic analysis step described in Section 2.1.2 is not employed; words are treated as pure lexical units, that is, they have no *meaning* associated with them, and sentences are treated as pure syntactic constructs. The semantic representations that are generated represent only the syntactic categories of the basic constituents.

Definite Clause Grammars

Definite clause grammars, or DCG's, were developed by Fernando C. N. Pereira and David H. D. Warren and presented in [14]. DCG's constitute a subset of the set of so-called *logic grammars*, which are directly inspired by the formalisms and techniques of predicate logic: the grammar rules of a DCG have the form of predicates which allow arguments, and thus context dependent information, to be passed between them.

Analysis

The analyzer of *NaSh* is based on a definite clause grammar for language analysis presented by Pereira and Warren in [14].

The basic idea behind the approach taken, is the application of the principle of *compositionality*, that is, the semantic representation of any constituent is composed from the semantic representation of its component parts. The computation of the semantic representation of a sentence therefore relies on

- The representation of individual words, given in the dictionary. At this level, variables are introduced as well.
- The rules describing how the semantic representation of a constituent is composed from the semantic representations of its constituent parts.

For a detailed description of the mechanisms behind the analyzer, see for instance [6], [13] or [14]. The logical representation scheme employed in *NaSh*, *NIKS*, is described in Section 2.3.1.

Generation

As mentioned in Section 3.1, the main goal with *NaSh* has been that its user interface should be exclusively based on *SNaL*. That is, knowledge passing, both from the user to the system and from the system to the user, should be done in *SNaL*. Since *NaSh* uses *NIKS*, not *SNaL*, when it reasons, it is necessary that the knowledge be paraphrased from *NIKS* into *SNaL* before it is presented to the user. This task is of nature not trivial, and is known as *automatic text generation* (see [6]).

The approach taken in *NaSh* is a rather naïve one, and his heavily dependent on the structure of *NIKS*, but it works fairly well, and is easy to understand, and thus also to modify. The cornerstone of the approach is a DCG for paraphrasing basic *NIKS*-structures into simple sentences. Deeply nested structures are treated recursively in a depth-first manner, and other complex structures are decomposed into their components, which are then treated separately. The simple sentences which result from this process may be conjugated and in other ways combined to form the final *SNaL*-representation. Also, depending on whether the structure being paraphrased represents a statement or a query, different approaches are taken. The approach used in making a statement out of the resulting sentences is fairly straight-forward, based on conjugation and nesting, whereas the approach for making queries is more complex, as queries have an inherently different structure from statements.

Some examples of *NIKS*-structures and the *SNaL*-representations resulting from the paraphrasing are shown below.

`man(john)` [query]	$\Rightarrow$	*Is John a man?*
`man(john)`	$\Rightarrow$	*John is a man.*
`in(with(live(john),mary),london)` [query]	$\Rightarrow$	*Does John live with Mary in London?*
`in(with(live(john),mary),london)`	$\Rightarrow$	*John lives with Mary in London.*
`which(X):(place(X)&in(live(john),X))`	$\Rightarrow$	*Where does John live?*
`which(X):(person(X)&of(mother(X),mary))`	$\Rightarrow$	*Who is a mother of Mary?*
`exists(X):((ship(X)&old(A))&see(john,A))`	$\Rightarrow$	*John sees an old ship.*

3.2.4 Limitations

When considering *SNaL*, it is important to keep in mind, that it is no more than a prototype, and can thus be expected neither to be sound nor complete. Some of the most evident limitations of *SNaL* are the following:

- No semantic analysis is done. This means that for instance the sentences "*Peter is a student in 1992*" and "*Gerard is the mayor of Auvers*" are identical as far as their semantical representations are concerned, in that the semantical representation of a sentence is determined exclusively by the basic constituents of which it is composed. As a consequence, no temporal information can be represented (as is evident from the example), thus seriously reducing the practical usefulness of the system.
- Of the language, only the present tense is handled. This only confirms what was stated above, that no temporal information can be represented.
- Only the third person singular is handled. This means that it is impossible to refer to groups of individuals other than by referring directly to the class to which they belong, which is hardly useful in general applications.
- In English, compound nouns are not formed by concatenating basic words; rather, they are represented as sequences of separate words. In *SNaL*, a noun has to be a single word, thus imposing serious restrictions on the set of legal nouns.
- Possessives are not handled, neither through possessive pronouns nor possessive suffixes. This imposes further restrictions on the usefulness of the system.
- Queries containing dangling prepositions, like in "*Who does Sarah play with?*", are not handled at present. This is not a major problem, though, in that it can be incorporated rather easily.
- As stated in Section 3.2.2, complex implicit rules may result in an unhandled case, that is, a logical representation containing more than one implication. This imposes a factor of indeterminism on the system, which ought not to be there.

3.3 The Expert System Shell

As mentioned in Section 3.1, the aim of *NaSh* has been to make a working prototype of a system containing the basic features of an expert system shell, and of which the dialogue is entirely based on *SNaL*. *SNaL*, both the language and the analyzer, have been thoroughly treated in Section 3.2. In this section, the reasoning system employed in *NaSh* is described.

3.3.1 Asserting Facts and Rules

The internal database of *NaSh* contains two categories of knowledge: *facts*, that is, declarative statements about the world, and *rules*.

After a sentence supplied by the user has been transformed into the corresponding *NIKS*-representation by the analyzer, this representation is passed on to the *evaluator*, which, depending on the category the sentence belongs to (statement, rule or query) decides what to do next. In this section we look at how statements and rules are treated by the evaluator, whereas the treatment of queries is described separately in Section 3.3.2.

Asserting Rules

Asserting rules is straight-forward in *NaSh*: they are explicitly stored in the knowledge base in their given form without any further treatment.

Asserting Facts

Asserting facts could have been treated similarly to the assertion of rules, that is, to be stored explicitly without further delay. But, since facts as supplied by the analyzer may contain large amounts of implicit knowledge which may be cumbersome to extract later as part of the reasoning process, I found it convenient to try to extract as much of this knowledge as possible at an as early stage as possible, and to store each piece of knowledge thus extracted separately.

To illustrate the approach, consider the sentence

John lives in a beautiful house in France

As is evident, this sentence indeed conveys that John lives in a beautiful house in France, but it also conveys a lot more. For instance that John lives in France, or that there is at least one beautiful house in France. To extract this knowledge, we can use the fact that it is contained in *nested prepositional phrases* to recursively go through the semantic structure representing the sentence, collecting noun phrases (like, in this case, John and the beautiful house) and combining them with the prepositional phrases from whose scope they are visible. Thus it is extracted that John lives in France, but not that France is in a beautiful house. To complete the example, these are the pieces of knowledge extracted from the given sentence by *NaSh*, presented in paraphrased form:

John lives in a beautiful house in France.
John lives in a beautiful house.
A beautiful house is in France.
John lives in France.
John lives.

From a mere syntactic point of view, as that of *NaSh*, this is about all that is possible to deduce about John and his beautiful house in France. Knowledge made up from nouns and their qualifiers is not extracted. This is because the combinations of nouns and qualifiers are considered to be separate entities, not variants of the same entity. Thus, in the above example, it is not deduced that "*A house is beautiful*".

It has to be pointed out, though, that the algorithm is not robust, in that it misses pieces of knowledge contained in certain complex nested constructs.

3.3.2 Answering to Queries

In the following discussion, the term *literal* means a constituent part of a conjugation, and the verb to *solve* is used to denote the process of proving a given literal.

The reasoning mechanism of *NaSh* works analogously to the resolution principle described in Section 2.3.2. Whereas resolution requires that the clause to be solved be a disjunction of literals, the mechanism employed in *NaSh* requires that the clause be a conjunction of literals. Moreover, implications (rules, that is) need not be converted to the analogue of clause form.

To supply an answer to a query, it is decomposed into a set of literals, where each literal has to be solved separately. Whenever one of the literals in the set has been solved, it is removed from the set, and thus the entire query being solved corresponds to each of its constituent literals being solved (this is analogous to a conjunction being true only if all of its constituents are true).

Simple literals may be solved in a number of ways:

- If the literal contains no variables and is contained in the knowledge base, or a literal is contained in the knowledge base which can be unified with the given literal, this literal has been solved.
- If a rule is found whose right side can be unified with the given literal, the rule is fired, which means that the left side of the rule is substituted for the literal, and thus has to be solved in order for the literal to be solved.

- If a rule is found whose right side constitutes is a conjugation of literals, one of which can be unified with the given literal, the rule is fired, and the process continues as above[1].
- If the leftmost argument of a prepositional literal which cannot be solved as it is, is constituted by a non-atomic literal, an attempt is made at solving this non-atomic literal. If the attempt succeeds, the literal is recombined with the enclosing expression, and a new attempt is made at solving it.
- If neither of the above stages have caused the literal to be solved, the user is prompted for the information needed for a solution to be arrived at. If the information thus acquired is insufficient, failure is reported and the search process terminates.

If the query has been solved, the answer it supplies depends on the nature of the query. That is, a confirmative query (see Section 3.2.2) supplies a "*yes*" or "*no*" answer, whereas all other queries (excluding "*how*"-queries) corresponds to the user wanting to know the *name* of a certain individual (instance) of a class of world objects, and thus this name constitutes the answer given.

3.3.3 Collecting Proofs

Proofs are collected during the reasoning process. Each time a literal has been solved, or a rule has been fired, it (the solved literal or the rule) is added to a list which is passed along through each step of the reasoning process. When the initial query has been solved, this list constitutes a proof as to how the conclusion was arrived at, and will be presented to the user on request (that is, if the query was prefixed with the keyword "*how*", see Section 3.2.2).

3.3.4 Cooperativity Issues (Not Implemented)

As mentioned in Section 2.1.3, Grice' Cooperation Principle has become widely accepted as a model on which to base intelligent (cooperative) interfaces to NLP-systems. Several approaches have been proposed on how to implement cooperative interfaces. In this section, two of these approaches, *integrity constraints* and *scripts analysis*, are briefly touched upon. In *NaSh*, no measures have been taken in order to make its interface cooperative.

Integrity Constraints

In [6], an approach is presented which is based on detecting disparities between the knowledge base (or database) and the assumptions the user makes about it.

Different users of the database may have different perceptions (or *viewpoints*) of it, which can be described by a set of *intensional* laws (that is, laws used to derive new facts from what is explicitly stored in the database, such as "*1848 is before 1989*". These laws can also be used to express semantic restrictions, or *integrity constraints* associated with the database. An example of an integrity constraint would be that one person cannot at the same time both be a mother and a father.

The *presupposition* of a query can be loosely defined as all the more general statements resulting from the query. For example, the query "*How many people took the train to Haugesund in 1991?*" presupposes that there exists a railway connection to Haugesund. When a very general presupposition is false, all the presuppositions less general than it are false also.

A *misconception* occurs between a user and a database when no answer to the user's query exists because either a presupposition of the query is found to be false, or one ore more integrity constraints are violated. Several misconception may occur in connection with the same query. In [6], several rules are listed to guide the system in the selection of which information to present to the user as a response to a query containing one or more misconceptions.

For a thorough description of the approach, as well as a Prolog implementation of parts of it, see [6].

[1]This is logically justified as follows. The given rule is expressed as $P \rightarrow Q_1 \wedge Q_2 \wedge \cdots \wedge Q_i \wedge \cdots \wedge Q_n$, which is equivalent to $(\neg P \vee Q_1) \wedge (\neg P \vee Q_2) \wedge \cdots \wedge (\neg P \vee Q_i) \wedge \cdots \wedge (\neg P \vee Q_n)$. In other words, $\neg P \vee Q_i$, which is equivalent to $P \rightarrow Q_i$, logically follows from the initial rule.

Scripts Analysis

A *script* may be defined as a stereotyped description of a situation which may occur within a certain domain, say, the act of going to a restaurant, where each entity is assigned a particular rôle, the waiters serving food, the guests entering, ordering, eating, paying and leaving. A script consists of a set of *slots*, which represent people and objects that are involved in the events described in the script. A script has associated with it a *header*, which describes the prerequisites for its being activated, as well as its preferred locations, rôles and events. One of the most important features of scripts, is their ability to predict events which have not been explicitly observed.

The main use of scripts has been in the field of *plan recognition*, but they have inherent features, such as their ability to detect the most likely course of events, which make them well suited as a basis for cooperative interfaces. For a more thorough treatment of scripts, see for instance [11] and [15].

3.3.5 Limitations

When evaluating *NaSh*, it is important to keep in mind that it, like *SNaL*, is no more than a prototype, and should thus be expected neither to be complete nor sound. Many of its main limitations are of an implementational nature, in that the system as of yet is not fully implemented. Neither has it undergone thorough and error-seeking testing. Rather, its test set has been a limited set of rules and declarative statements, and errors have been corrected which were discovered using this set. Some of its limitations can be summarized as follows (see also Section 3.2.4):

- Each time the system is started, its knowledge base is empty. This means that the user has to, by hand, supply it with the necessary rules and facts. This is not satisfactory under any circumstances, and particularly not when the set of rules and facts constituting the given domain becomes large.
- The user may not ask *why* he is asked a question. It is desirable that he be able to do so, as a means to establish his confidence in the system.
- When the system has arrived at an answer to a query, it is presented as a single-word response, not as a complete sentence. This is not a major issue, though, but it would be desirable from the user's point of view that full-sentence responses be given.
- The set of separate pieces of knowledge derived from a declarative statement is not complete (see Section 3.3.1).
- When trying to prove a query on the basis of a set of rules, the system fires a rule and then loops forever if it is not able to prove the query given that particular rule. This is caused by the reasoning mechanism not being equipped with a counter ensuring that failure eventually is reported if the set of unsolved literals does not become empty (see Section 3.3.2). A minor bug to fix, though.
- The system is not reliable, neither is its behaviour predictable. Depending on the structure of the constructs representing the query to be solved, the system may or may not arrive at a solution. Most of the time it does, though (arrive at a solution, that is).

There are other limitations as well, mostly due to implementational incompleteness, which are not elaborated upon here.

3.4 An Example Domain - The Norwegian Nationality Act

As an example domain for *NaSh*, I have chosen the Norwegian Nationality Act. This is due to the fact that legislation, being basically definitional in nature, is particularly suited for formalization. This example is presented as though *NaSh* is working completely in accordance with its specification. For the most part this is true, at least for the given domain, but no guarantee is given that the system will behave well in all circumstances (see Section 3.3.5).

3.4.1 User Commands

In order for the user to be able to ask the system to perform specific tasks not directly connected to the reasoning process, *NaSh* is equipped with a (small) set of user commands. These are

- *summary*: This command causes the system to print the contents of the knowledge base, paraphrased into *SNaL*. Facts are preceded with the keyword "*known:*", and rules with the keyword "*rule:*".
- *resk*: Retract all facts from the knowledge base.
- *resr*: Retract all rules from the knowledge base.
- *reset*: Retract everything from the knowledge base, both facts and rules.

In addition the two built-in predicates *debug* and *nodebug* may be invoked from *NaSh*.

3.4.2 Judicial Expert Systems

In judicial expert systems, the legislation is the primary source of the knowledge base. In addition, it may be influenced by other legislative factors, such as *case law* (law established by judicial decision in cases) and judicial literature. An expert system with the Norwegian Nationality Act as its domain will be able to answer questions like "*Can Eigil Skallagrimsson become a Norwegian citizen?*".

It is important to have in mind that judicial expert systems never can replace a judicial system. Legislation often contains *vague* statements, such as "*being of good character*" and "*having sufficient knowledge of Norwegian*", as well as other ambiguities, and sometimes even contradictions. This demands careful consideration in each case to which the particular piece of legislation is applied, consideration which an automated reasoning system such as an expert system will not be able to support, at least not in the foreseeable future. The main task of judicial expert systems is thus a supervisory one; they may act as a guide in finding one's way through intricate law texts, and may point to areas demanding extra consideration.

The pioneer project in the field of legislative expert systems was a system based on the British Nationality Act, done at the Imperial College, U. K., and presented in [16]. The Norwegian Nationality Act has been used previously as domain for judicial expert systems in connection with the *HSQL*-project at NTH; see for instance [3] and [10].

3.4.3 Creating the Knowledge Base

In Appendix B, the extract from the Norwegian Nationality Act on which the knowledge base of the example is based, is given.

In order for the system to be able to understand the text, it has to be translated into *SNaL*, in the form of formal rules. That is, the paragraphs have to be transformed into *if-then*-constructs, where the premises are conjugations of simple declarative statements, and the conclusions are single declarative statements. The given paragraphs have already been logically formalized in [5], and are thus easily translated into *SNaL*. In this example we will concentrate on §1, which gets the following representation:

```
If any woman is a citizen in any year and
   this woman is the mother of any person and
   this person is born in this year
then
   this person is a citizen by birth.

If any man is a citizen in any year and
   this man is the father of any person and
   any woman is the mother of this person and
   this man marries this woman in any other year and
```

this other year is before this year and
this person is born in this year
then
this person is a citizen by birth.

If any man is the father of any person and
any woman is the mother of this person and
this man dies in any year and
this man is a citizen in this year and
this man is married to this woman in this year
then
this person is a citizen by birth.

If any person is an abandoned orphan
then
this person is a citizen.

"*Citizen*" in this context is to be interpreted as "*Norwegian citizen*". These formalized rules can be fed directly to *NaSh*, which stores them as formal rules in the knowledge base.

3.4.4 A Sample Dialogue

If we want to find out whether a certain person, e.g., Kristin, is a Norwegian citizen, we ask the question "*Is Kristin a citizen?*" to the system. In order for the system to know that any citizen by birth also is a citizen, we have to supply it with the additional rule (here given as an implicit rule):

Every citizen by birth is a citizen.

The system will now try to prove that Kristin indeed *is* a citizen, first by investigating whether this is an already known fact (which we assume it is not in our case), then by applying the above rules one by one until either a proof has been found or failure is reported. When applying a certain rule, the system may have to query the user for information in order to arrive at a proof.

Interacting with the User

The first assertion that the system has to prove (the initial question, that is), is whether Kristin is a citizen. There exist two rules in the knowledge base describing how she can be classified as a citizen. One says that she is a citizen if she is an abandoned orphan, the other that she is a citizen if she is a citizen by birth. The rules are applied in order of appearance in the knowledge base, and thus the first question asked to the user is

Is Kristin an abandoned orphan?

If the answer given is "*yes*", the assertion has been proved, and the search process terminates. If the answer is "*no*", the system tries to prove that Kristin is a citizen by birth, to which it can apply either of the three first rules above. The questions asked in connection with each rule, as well as answers causing a solution for the particular rule, are:

When is Kristin born? Kristin is born in 1966.
Who is a mother of Kristin? Sigrid.
Is Mary a citizen in 1966? Yes.

When is Kristin born? 1966.
Who is a mother of Kristin? Sigrid is the mother of Kristin.
Who is a father of Kristin? Jammælt.
Is Jammælt a citizen in 1966? Yes.
When does Jammælt marry Sigrid? Jammælt marries Sigrid in 1964.

> *Who is a mother of Kristin?* Sigrid.
> *Who is a father of Kristin?* Jammælt.
> *When does Jammælt die?* 1964.
> *Is Jammælt a citizen in 1964?* Yes.
> *Is Jammælt married to Sigrid in 1964?* Yes.

If the information supplied by the user does not suffice in proving that Kristin is a citizen by any rule in the knowledge base, the user is notified that failure has occurred.

Note that it is possible to give full-sentence answers as well as one word answers. In fact, it is possible to supply answers which may have no relevance whatsoever with the question. The information contained in these answers is added to the knowledge base and may be used to answer later questions, which thus do not have to be presented to the user. For instance, if given the question "*Who is a mother of Kristin?*", the answer may well be "*Jammælt dies in 1964*". The system still does not know who Kristin's mother is, and will have to ask the question again, but it *does* know that Jammælt dies in 1964. In other words, all questions given in connection with a specific rule have to be answered in order to arrive at a conclusion, but the order in which the answers are given does not have to correspond with the order of the questions.

Supplying Proofs

If the user wants a proof as to why Kristin is a citizen, he asks the question "*How is Kristin a citizen?*" to the system. For instance, if the system is able to prove that Kristin is a citizen by the second rule above, the proof supplied to the user may be (how proofs are collected is described in section 3.3.3):

> *Kristin is a citizen if Kristin is a citizen by birth.*
> *Kristin is a citizen by birth if*
>   *A is a man and B is a year and A is a citizen in B and*
>   *A is a father of Kristin and*
>   *C is a woman and C is a mother of Kristin and*
>   *D is a year and A marries C in D and D is before B*
>   *and Kristin is born in B.*
>
> *1964 is before 1966.*
> *Jammælt is a citizen in 1966.*
> *Jammælt is a father of Kristin.*
> *Sigrid is a mother of Kristin.*
> *Jammælt marries Sigrid in 1964.*
> *Kristin is born in 1966.*

Summaries

In addition to query the system and ask for proofs, the user may ask to be given a summary of the contents of the knowledge base. The command *summary* is used for this purpose. For instance, if the knowledge base has the same contents as that of the previous example, the summary given will look like:

> *Known: Kristin is born in 1966.*
> *Known: Kristin is born.*
> *Known: Sigrid is a mother of Kristin.*
> *Known: Sigrid is a mother.*
> *Known: Jammælt is a father of Kristin.*
> *Known: Jammælt is a father.*
> *Known: Jammælt is a citizen in 1966.*
> *Known: Jammælt is a citizen.*
> *Known: Jammælt marries Sigrid in 1964.*
> *Known: Jammælt marries Sigrid.*

Known: Sigrid is in 1964.
Known: Jammælt is in 1964.

Rule: C is a citizen by birth if
  A is a man and B is a year and A is a citizen in B and
  C is a person and A is a father of C and
  D is a woman and D is a mother of C and
  E is a year and A marries D in E and E is before B and
  C is born in B.
Rule: A is a citizen if
  A is a person and A is a citizen by birth

Note that some of the information contained in the knowledge base is inconsistent with common sense knowledge about the world. For instance, it is not sound to deduce from the fact that Jammælt is (or rather, *was*) a citizen in 1966 that he still is a citizen, which is how the average user would interpret the sentence "*Jammælt is a citizen*". This kind of inconsistence is a consequence of *NaSh*' lack of semantic analysis (see Sections 3.2.3 and 3.2.4), and may cause the knowledge base to contain some rather awkward pieces of knowledge.

3.5 Future Extensions

The limitations of *SNaL* and *NaSh* are elaborated upon in Sections 3.2.4 and 3.3.5. Some possible future extensions for the system as a whole are the following:

- Increase the reasoning powers of the system, by including semantic analysis. In addition, it would be desirable to equip it with mechanisms supporting temporal reasoning, which is a major requirement that has to be fulfilled if the aim of the system is that it be of practical use.
- Extend the definition of *SNaL* also to handle the past tense, other inflections than the third person singular, compound nouns, possessives and dangling prepositions.
- Provide the system with an automated input mechanism for asserting rules and facts when it is initialized. For instance, the system may be initialized with a file as input parameter, containing the required rules and facts.
- The bug causing the system to loop forever when the chosen rule does not apply, has to be fixed. How this can be done, is described in Section 3.3.5.

Generally, it would be desirable to improve both the definition of the language as well as the reasoning mechanisms involved, in order to make the system of practical as well as theoretical interest. An approach to improving the reasoning mechanism, would be to base it on Tore Amble's powerful logical representation scheme *SOLON*, rather than *NIKS*.

Chapter 4

Conclusion

In working with this project, my aim has been to investigate some of the problems pertaining to the field of natural language processing, and to develop a suitable platform on which to base the implementation of a natural language based expert system shell.

4.1 Evaluation

The implementation, as of today, is by no means complete, as discussed in previous chapters. As one of the prerequisites of my work has been that it should take place within the framework of *TUC*, it is of importance that the system serves a purpose within this framework. Whereas one of the goals with *TUC* is that it shall demonstrate semantic as well as temporal features, my implementation contains no such features. This is obviously a drawback. On the other hand, *TUC* is very restrictive with respect to the *meanings* assigned to particular words, in that the notion of *situations* is defined in terms of them. This does not correspond very well with natural languages as used in daily life, where different words assume different meanings depending on the context in which they appear. In my implementation, words have no meanings associated with them, and may thus be used differently in different contexts. A desirable solution would be a combination of these two approaches, featuring both the semantic powers of *TUC* as well as the expressive powers resulting from flexible use of words. Thus I hope that my system will serve a purpose as a contributor to the continued development of *TUC*.

4.2 Summary

In working with this project, I have had the opportunity to delve deeper into what I find to be one of the most interesting as well as challenging fields of computer science. I have found out that my *a priori* interest in the field of natural languages and natural language processing indeed was more than a mere superficial fancy, and I hope that I in the future will get the opportunity to continue working with some of the questions I found interesting during the work.

Appendix A

Definition of a Semi-Natural Language

In what follows, a definition, represented in BNF, of the semi-natural language *SNaL* is given. Since BNF is only able to represent context-free grammars, the context-dependent information contained in the DCG-implementation of the grammar has been lost. On the other hand, the BNF-representation is easier to read than the DCG-implementation, and is thus given here.

The start symbol is *sentence*, non-terminals are printed using a *slanted* typestyle, and terminals are in **bold**. The terminals have associated with them a set of lexical insertion rules (see Section 3.2.1), which is not shown here. Productions are presented in a depth-first manner. ϵ

represents the empty string.

sentence	→	*statement*
	|	*query*
statement	→	*decl_statement*
	|	*formal_rule*
decl_statement	→	*simple_decl_statement*, [*and*], *decl_statement*
	|	*simple_decl_statement*
simple_decl_statement	→	*noun_phrase*, *verb_phrase*
noun_phrase	→	*determiner*, *adj_phrase*, **noun**, *mod_rel*
	|	*atomic_noun_phrase*
adj_phrase	→	**adj**, *adj_phrase*
	|	**adj**
	|	ϵ
mod_rel	→	*modifier*
	|	*relative_clause*
modifier	→	**modifier_prep**, **determiner**, *adj_phrase*, **noun**, *relative_clause*
	|	**modifier_prep**, *atomic_noun_phrase*
relative_clause	→	**rel_pronoun**, *verb_phrase*
	|	ϵ
atomic_noun_phrase	→	**name**
	|	**abstract_noun**
	|	**year**
	|	**var_noun**
verb_phrase	→	*simple_verb_phrase*, *prep_phrases*
simple_verb_phrase	→	**intrans_verb**
	|	**trans_verb**, *noun_phrase*
	|	**aux(be)**, *aux_tail*
	|	*trace_verb_phrase*
prep_phrases	→	*prep_phrase*, *prep_phrases*
	|	*prep_phrase*
	|	ϵ
prep_phrase	→	**prep**, *noun_phrase*
aux_tail	→	*comp*
	|	*prep_phrase*
comp	→	**adj**
	|	*noun_phrase*
trace_verb_phrase	→	*noun_phrase*, **trans_verb**
formal_rule	→	[*if*], *decl_statement*, [*then*], *decl_statement*
query	→	*is_does_phrase*
	|	**wh_pronoun**, *vp_aux*
is_does_phrase	→	*is_phrase*
	|	*does_phrase*
is_phrase	→	**aux(be)**, *noun_phrase*, *comp_prep_phrase*, *prep_phrases*
does_phrase	→	**aux(do)**, *simple_decl_statement*
comp_prep_phrase	→	*comp*
	|	*prep_phrase*
vp_aux	→	*verb_phrase*
	|	*aux_phrase*
aux_phrase	→	**aux(do)**, *trace_simple*
	|	**aux(be)**, *noun_phrase*, *aux_tail*, *prep_phrases*
trace_simple	→	*trace_verb_phrase*
	|	*simple_decl_statement*

Appendix B

Extract from the Norwegian Nationality Act

Below the first three paragraphs of the Norwegian Nationality Act (Dec. 8., 1950) are quoted, in Norwegian (footnotes are omitted):

Kapitel 1. Korleis folk får borgarretten

§1. Barn får norsk borgarrett når det kjem til:

a) dersom mora er norsk borgar,
b) dersom faren er norsk borgar og foreldra er gifte,
c) dersom faren er død, men var norsk borgar og gift med mor til barnet då han døydde.

Hittebarn som er funne her i riket, vert rekna for norsk borgar til dess noko anna vert opplyst.

§2. Har norsk mann og utenlandsk kvinne barn i lag frå før dei gifter seg med kvarandre, får barnet norsk borgarrett når foreldra vert vigde, så framt det er ugift og under 18 år.

§3. Utlending som har budd i riket frå fylte 16 år og tidlegare samanlagt i minst 5 år, får norsk borgarrett når han etterat han fylte 21 år, men før han fyller 23 år, gjev inn skriftleg melding til fylkesmannen om at han vil vera norsk borgar. Har han ikkje borgarrett i noko land, kan han gjeve inn slik melding så snart han har fylt 18 år, dersom han når han gjev inn meldinga har hatt bustad i riket dei siste 5 åra og tidlegare har budd her i minst 5 år til; det same gjeld såframt han etterviser at han misser den framande borgarretten når han får norsk borgarrett.

Er Noreg i krig, kan ingen borgar i fiendestat få norsk borgarrett etter denne paragrafen. Det same gjeld den som er borgarrettslaus, men som seinast hadde borgarrett i fiendestat.

Appendix C

Source Files

In this appendix a listing of the source files making up *NaSh* is given in alphabetical order. The main program is contained in the file *citizen.pl*.

aux.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%% -*- Mode: Prolog -*- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% aux.pl ---
%%% Author          : Anders Blehr
%%% Created On      : Fri Apr  3 12:24:27 1992
%%% Last Modified By: Anders Blehr
%%% Last Modified On: Sun May  3 11:28:30 1992
%%% RCS revision    : $Revision: 1.19 $ $Locker: blehr $
%%% Status          : Under development
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

forget(X) :-
        retract(X),
        false;
        true.

reset :-
        nl,
        resk,
        resr,
        nl.

resk :-
        nl,
        forget(known(_)),
        write('known reset'),
        nl.

resr :-
        nl,
        forget(rule(_)),
        write('rules reset'),
        nl.

summary :-
        nl,
        write_facts,
        nl,
        write_rules.

write_facts :-
        known(X),
        numbervars(X,0,_),
        write('known: '),
        paraphrase(noquery,X),
        nl,
        false;
        true.

write_rules :-
        rule(X),
        numbervars(X,0,_),
        write('rule: '),
        paraphrase(rule,X),
        nl,
        false;
        true.

write_proof([]).
write_proof(Proof) :-
        pop(Prem=>Concl,Proof,Proof1),
        paraphrase(rule,Prem=>Concl),
        nl,
        nl,
        write_proof(Proof1).

write_proof(Proof) :-
        pop(Subproof,Proof,Proof1),
        paraphrase(noquery,Subproof),
        nl,
        nl,
        write_proof(Proof1).

%%

append([],X,X).
append([X|Y],Z,[X|W]) :- append(Y,Z,W).

pop(X,List,NewList) :- append([X],NewList,List).

member(X,[X|_]).
member(X,[_|Tail]) :- member(X,Tail).

reverse(List,RevList) :- rev(List,[],[],RevList).

rev([],[],RevList,RevList).
rev(List,NList,RevList,NRevList) :-
        pop(Elt,List,TmpList),
        append([Elt],RevList,TmpRevList),
        rev(TmpList,NList,TmpRevList,NRevList).
```

```
listify(exists(X):(Rep),ListRep) :-
        listify(Rep,RepList),
        append([exists,'(',X,')',':','('],RepList,TmpList),
        append(TmpList,[')'],ListRep).

listify(which(X):(Rep),ListRep) :-
        listify(Rep,RepList),
        append([which,'(',X,')',':','('],RepList,TmpList),
        append(TmpList,[')'],ListRep).

listify(Rep1&Rep2,ListRep) :-
        listify(Rep1,Rep1List),
        listify(Rep2,Rep2List),
        append(Rep1List,[&],TmpList),
        append(TmpList,Rep2List,ListRep).

listify(Rep,[Rep]) :- atom_term(Rep).

listify(Rep,[P,'(',Q,')']) :-
        Rep=..[P,Q],
        \+ (P='$VAR').

listify(Rep,ListRep) :-
        Rep=..[P,Q,R],
        listify(Q,QList),
        append([P,'('],QList,TmpList),
        append(TmpList,[',',R,')'],ListRep).

unify(X,X).

get_noun(Rang,Noun,Vars,X) :- pop((Rang,Noun,X),Vars,_),!.
get_noun(Rang,Noun,Vars,X) :-
        \+ (Vars = []),
        pop(_,Vars,Vars1),
        get_noun(Rang,Noun,Vars1,X).

get_article(Noun,an) :-
        convert(Noun,NList),
        pop(C,NList,_),
        vowel(C).

get_article(_,a).

infinitive(Inf,Pres) :- inf(Inf,[C,y],not vowel(C),[C,i,e,s],Pres).
infinitive(Inf,Pres) :- inf(Inf,[C],member(C,[o,s,x,z]),[C,e,s],Pres).
infinitive(Inf,Pres) :- inf(Inf,[C,h],member(C,[c,s]),[C,h,e,s],Pres).
infinitive(Inf,Pres) :- inf(Inf,[],true,[s],Pres).

present(Inf,Pres) :- pres(Inf,[C,y],not vowel(C),[C,i,e,s],Pres).
present(Inf,Pres) :- pres(Inf,[C],member(C,[o,s,x,z]),[C,e,s],Pres).
present(Inf,Pres) :- pres(Inf,[C,h],member(C,[c,s]),[C,h,e,s],Pres).
present(Inf,Pres) :- pres(Inf,[],true,[s],Pres).

inf(Inf,Astring,Condition,Bstring,Pres) :-
        convert(Pres,Preslist),
        append(Base,Bstring,Preslist),
        call(Condition),
        append(Base,Astring,Inflist),
        convert(Inf,Inflist).

pres(Inf,Astring,Condition,Bstring,Pres) :-
        convert(Inf,Inflist),
        append(Base,Astring,Inflist),
        call(Condition),
        append(Base,Bstring,Preslist),
        convert(Pres,Preslist).

convert(X,X1) :-
        var(X1),!,
        name(X,Codes),
        name1(Codes,X1).

convert(X,X1) :-
        var(X),!,
        name1(Codes,X1),
        name(X,Codes).

name1([],[]).
name1([X|Xs],[X1|X1s]) :-
        name(X1,[X]),
        name1(Xs,X1s).

%%

vowel(a).
vowel(e).
vowel(i).
vowel(o).
vowel(u).
```

citizen.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%% -*- Mode: Prolog -*- %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% citizen.pl ---
%%% Author          : Anders Blehr
%%% Created On      : Sat Apr  4 13:47:05 1992
%%% Last Modified By: Anders Blehr
%%% Last Modified On: Tue Apr 28 14:45:53 1992
%%% RCS revision    : $Revision: 1.19 $ $Locker: blehr $
%%% Status          : Under development
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- consult(grammar).     % grammar
:- consult(aux).         % auxiliary predicates
:- consult(database).    % database
:- consult(dictionary).  % dictionary
:- consult(para).        % paraphrase from LR to NL
:- consult(readin).      % converts text to list format
:- consult(remember).    % asserting rules and extracted facts
:- consult(solve).       % expert system shell

:- dynamic known/1.
:- dynamic rule/1.

run :-
        repeat,
        nl,
        nl,
        write('NRL: '),
        read_sentence(user,List),
        (List=[stop|_];
            (List=[C|_],
             member(C,[debug,nodebug,reset,resk,resr,summary]),
             call(C),
             run);
            (sentence(subject(X),Rep,Type,List,[]),
             prettyprint(Rep),
             evaluate(Type,subject(X),Rep),
             nl,
             run)).

evaluate(query,_,Rep) :- evaluate(Rep).
evaluate(stmt,subject(X),Rep) :- rem(subject(X),Rep).

evaluate(how:Rep1) :- !,
        nl,
        remove_instQs(Rep1,Rep),
        solve(Rep,[],_,[],Proof),
        nl,
        write_proof(Proof),
        nl,
        write('yes'),
        false;
        true,
        nl.
```

```
evaluate(which(X):Rep1) :- !,
        nl,
        remove_instQs(which(X):Rep1,Rep),
        which_solve(Rep,[],_),
        write(X),
        nl,
        false;
        true,
        nl.

evaluate(Rep1) :- !,
        nl,
        remove_instQs(Rep1,Rep),
        solve(Rep,[],_,[],_),
        nl,
        write('yes'),
        nl,
        false;
        true,
        nl.

evaluate(_) :-
        nl,
        write('error'),
        nl,
        false;
        true,
        nl.

prettyprint(Rep) :-
        nl,
        nl,
        numbervars(Rep,0,_),
        write(Rep),
        nl,
        nl,
        false; % release bindings
        true.


exists(_):P :- P.

P=>Q :-not(P&(not Q)).

forall(_):(P=>Q) :- not(P&(not Q)).

X&Y :-
        known(Y),
        known(X).

not X :- \+ X.
```

database.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%% -*- Mode: Prolog -*- %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% database.pl ---
%%% Author          : Anders Blehr
%%% Created On      : Fri Apr  3 12:23:53 1992
%%% Last Modified By: Anders Blehr
%%% Last Modified On: Sun May  3 11:28:53 1992
%%% RCS revision    : $Revision: 1.9 $ $Locker: blehr $
%%% Status          : Under development
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Rules

person(X) :- man(X).
person(X) :- woman(X).

name(X) :- man(X).
name(X) :- woman(X).
name(X) :- place(X).

place(X) :- country(X).
place(X) :- city(X).

dead(X) :- not lives(X).

year(X) :-
number(X),
X >= 1900,
X =< 2000.

time(X) :- year(X).

% Facts (nouns)

man(ali).
man(anders).
man(haakon).
man(john).
man(ola).
man(otto).
man(peter).

woman(barbro).
woman(brit).
woman(chakra).
woman(gunhild).
woman(jane).
woman(kari).
woman(kristin).
woman(mary).

city(bergen).
city(brussel).
city(london).
city(leuven).
city(paris).
city(rome).

country(belgium).
country(england).
country(france).
country(germany).
country(italy).
country(norway).

place(europe).

% Relations (verbs)

live(ali).
live(anders).
live(barbro).
live(brit).
live(gunhild).
live(haakon).
live(jane).
live(john).
live(kari).
live(kristin).
live(mary).
live(ola).
live(otto).
live(peter).
```

dictionary.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%% -*- Mode: Prolog -*- %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% dictionary.pl ---
%%% Author          : Anders Blehr
%%% Created On      : Fri Apr  3 12:22:53 1992
%%% Last Modified By: Anders Blehr
%%% Last Modified On: Sun May  3 11:29:14 1992
%%% RCS revision    : $Revision: 1.19 $ $Locker: blehr $
%%% Status          : Under development
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

noun(beach).
noun(box).
noun(child).
noun(citizen).
noun(city).
noun(country).
noun(daughter).
noun(date).
noun(father).
noun(house).
noun(husband).
noun(man).
noun(mother).
noun(orphan).
noun(person).
noun(sea).
noun(son).
noun(wife).
noun(woman).
noun(year).

abst_noun(birth).
abst_noun(love).
abst_noun(marriage).

trans_verb(love).
trans_verb(make).
trans_verb(marry).

intrans_verb(die).
intrans_verb(live).
intrans_verb(run).

adj(abandoned).
adj(alive).
adj(beautiful).
adj(big).
adj(blue).
adj(born).
adj(dead).
adj(french).
adj(german).
adj(green).
adj(handsome).
adj(happy).
adj(illegitimate).
adj(italian).
adj(married).
adj(newborn).
adj(norwegian).
adj(old).
adj(red).
adj(sunlit).
adj(tired).
adj(yellow).
adj(young).

prep(after).
prep(at).
prep(before).
prep(by).
prep(in).
prep(on).
prep(outside).
prep(of).
prep(to).
prep(with).

mod_prep(by).
mod_prep(of).
```

grammar.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%% -*- Mode: Prolog -*- %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% grammar.pl ---
%%% Author          : Anders Blehr
%%% Created On      : Fri Apr  3 12:23:27 1992
%%% Last Modified By: Anders Blehr
%%% Last Modified On: Sun May  3 11:29:56 1992
%%% RCS revision    : $Revision: 1.30 $ $Locker: blehr $
%%% Status          : Under development
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- op(900,xfx,=>).
:- op(800,xfy,&).
:- op(700,fx,not).
:- op(300,xfx,:).

sentence(subject(X),Rep,stmt) --> statement(subject(X),Rep),!.
sentence(_,Rep,query) --> query(Rep).

statement(_,Rep1=>Rep2) -->
        [if],
        body(_,Rep1,[],Vars),
        [then],
        body(_,Rep2,Vars,_),
        [.].

statement(subject(X),Rep) -->
        body(subject(X),Rep,[],_),
        [.].

query(how:Rep) -->
        [how],
        is_does_phrase(Rep).

query(Rep) -->
        wh_pron(X,Rep1,Rep),
        vp_aux(X,Rep1),
        [?],!.

query(Rep) --> is_does_phrase(Rep).

is_does_phrase(Rep) --> is_phrase(Rep).
is_does_phrase(Rep) --> does_phrase(Rep).

is_phrase(Rep) -->
        aux(be),
        noun_phrase(X,Rep1,Rep2,[],Vars1),
        (comp(X,Rep1);prep_phrase(X,Rep1,Vars1,Vars2)),
        prep_phrases(Rep2,Rep,Vars2,_),
        [?].

does_phrase(Rep) -->
        aux(do),
        simple_body(_,Rep,[],_),
        [?].
```

```
vp_aux(X,Rep) --> verb_phrase(X,Rep,[],_),!.
vp_aux(X,Rep) --> aux_phrase(X,Rep).

aux_phrase(X,Rep) -->
        aux(do),
        trace_vp(X,Rep,[],_).

aux_phrase(_,Rep) -->
        aux(do),
        simple_body(_,Rep,[],_).

aux_phrase(_,Rep) -->
        aux(be),
        noun_phrase(X,Rep1,Rep2,[],_),
        aux_tail(X,Rep1,[],_),
        prep_phrases(Rep2,Rep,[],_).

body(subject(X),Rep,Vars1,Vars) --> simple_body(subject(X),Rep,Vars1,Vars).
body(_,Rep,Vars1,Vars) --> conj_body(Rep,Vars1,Vars).

simple_body(subject(X),Rep,Vars1,Vars) -->
        noun_phrase(X,Rep1,Rep,Vars1,Vars2),
        verb_phrase(X,Rep1,Vars2,Vars).

conj_body(Rep1&Rep2,Vars1,Vars) -->
        simple_body(_,Rep1,Vars1,Vars2),
        [and],
        body(_,Rep2,Vars2,Vars).

noun_phrase(X,Rep1,Rep,Vars1,Vars) -->
        determiner(X,Rep2,Rep1,Rep,Vars1,Vars2),
        np_core(X,Rep2,Vars2,Vars).

noun_phrase(X,Rep,Rep,Vars,Vars) --> simple_np(X,Vars).

np_core(X,Rep,Vars1,Vars) -->
        adjs(X,Rep1,Rep),
        noun(X,Rep2,Vars1,Vars2),
        mod_rel(X,Rep2,Rep1,Vars2,Vars).

verb_phrase(X,Rep,Vars1,Vars) -->
        vp(X,Rep1,Vars1,Vars2),
        prep_phrases(Rep1,Rep,Vars2,Vars).

vp(X,Rep,Vars,Vars) --> intrans_verb(X,Rep).

vp(X,Rep,Vars1,Vars) -->
        trans_verb(X,Y,Rep1),  % love
        noun_phrase(Y,Rep1,Rep,Vars1,Vars). % mary

vp(X,Rep,Vars1,Vars) -->
        aux(be),
        aux_tail(X,Rep,Vars1,Vars).
```

```
vp(X,Rep,Vars1,Vars) --> trace_vp(X,Rep,Vars1,Vars).

aux_tail(X,Rep,Vars,Vars) --> comp(X,Rep).
aux_tail(X,Rep,Vars1,Vars) --> prep_phrase(X,Rep,Vars1,Vars).

trace_vp(Y,Rep,Vars1,Vars) -->
        noun_phrase(X,Rep1,Rep,Vars1,Vars), % mary
        trans_verb(X,Y,Rep1).  % love

mod_rel(_,Rep1,Rep,Vars1,Vars) --> modifier(Rep1,Rep,Vars1,Vars).
mod_rel(X,Rep1,Rep,Vars1,Vars) --> rel_clause(X,Rep1,Rep,Vars1,Vars).

prep_phrases(Rep1,Rep,Vars1,Vars) -->
        prep_phrase(Rep1,Rep2,Vars1,Vars2),
        prep_phrases(Rep2,Rep,Vars2,Vars).

prep_phrases(Rep,Rep,Vars,Vars) --> [].

prep_phrase(Rep1,Rep,Vars1,Vars) -->
        prep(Rep1,Y,Rep2),
        prep_np(Y,Rep2,Rep,Vars1,Vars).

modifier(Rep1,Rep,Vars1,Vars) -->
        mod_prep(Rep1,Y,Rep2),
        mod_np(Y,Rep2,Rep,Vars1,Vars).

mod_np(X,Rep1,Rep2,Vars1,Vars) -->
        prep_det(X,Rep3,Rep1,Rep2,Vars1,Vars2),
        adjs(X,Rep4,Rep3),
        noun(X,Rep5,Vars2,Vars3),
        rel_clause(X,Rep5,Rep4,Vars3,Vars).

mod_np(X,Rep,Rep,Vars,Vars) --> simple_np(X,Vars).

prep_np(X,Rep1,Rep2,Vars1,Vars) -->
        prep_det(X,Rep3,Rep1,Rep2,Vars1,Vars2),
        np_core(X,Rep3,Vars2,Vars).

prep_np(X,Rep,Rep,Vars,Vars) --> simple_np(X,Vars).

simple_np(X,_) --> name(X).
simple_np(X,_) --> year(X).
simple_np(X,_) --> abst_noun(X).
simple_np(X,Vars) --> [this,third,Noun],{get_noun(third,Noun,Vars,X)}.
simple_np(X,Vars) --> [this,other,Noun],{get_noun(other,Noun,Vars,X)}.
simple_np(X,Vars) --> [this,Noun],{get_noun(first,Noun,Vars,X)}.

prep_det(X,Rep1,Rep2,forall(X):(Rep1&Rep2),Vars,Vars) --> [every].
prep_det(X,Rep1,Rep2,exists(X):(Rep1&Rep2),Vars,Vars) -->
        [a] | [an] | [the] | [some].
prep_det(_,Rep,Rep,Vars,Vars) --> [].

prep_det(X,Rep1,Rep2,exists(X):(Rep1&Rep2),Vars1,Vars) --> [any,third],
        {append([(insert_noun,third)],Vars1,Vars)}.
prep_det(X,Rep1,Rep2,exists(X):(Rep1&Rep2),Vars1,Vars) --> [any,other],
        {append([(insert_noun,other)],Vars1,Vars)}.
prep_det(X,Rep1,Rep2,exists(X):(Rep1&Rep2),Vars1,Vars) --> [any],
        {append([(insert_noun,first)],Vars1,Vars)}.

adjs(X,Rep1,Rep3) -->
        adj(X,Rep2),
        adjs(X,Rep1&Rep2,Rep3).

adjs(_,Rep,Rep) --> [].

rel_clause(X,Rep1,Rep1&Rep2,Vars1,Vars) -->
        rel_pron,
        verb_phrase(X,Rep2,Vars1,Vars).

rel_clause(_,Rep,Rep,Vars,Vars) --> [] .

comp(X,Rep) --> adj(X,Rep).
comp(X,Rep) --> ([a] | [an] | [the]),
        np_core(X,Rep,[],_).

determiner(X,Rep1,Rep2,forall(X):(Rep1=>Rep2),Vars,Vars) --> [every].
determiner(X,Rep1,Rep2,exists(X):(Rep1&Rep2),Vars,Vars) --> [a] | [an] | [the].

determiner(X,Rep1,Rep2,exists(X):(Rep1&Rep2),Vars1,Vars) --> [any,third],
        {append([(insert_noun,third)],Vars1,Vars)}.
determiner(X,Rep1,Rep2,exists(X):(Rep1&Rep2),Vars1,Vars) --> [any,other],
        {append([(insert_noun,other)],Vars1,Vars)}.
determiner(X,Rep1,Rep2,exists(X):(Rep1&Rep2),Vars1,Vars) --> [any],
        {append([(insert_noun,first)],Vars1,Vars)}.

wh_pron(X,Rep,which(X):(time(X)&in(Rep,X))) --> [when].
wh_pron(X,Rep,which(X):(place(X)&in(Rep,X))) --> [where].
wh_pron(X,Rep,which(X):(person(X)&(Rep))) --> [who].
wh_pron(X,Rep,which(X):(Rep1&Rep)) --> [which],
        np_core(X,Rep1,[],_).

rel_pron --> [that] | [who] | [which].

aux(be) --> [is] | [are] | [was] | [were].

aux(do) --> [does] | [did].

noun(X,P,Vars1,Vars) --> [W],
        {noun(W),
         P=..[W,X],
         ((pop((insert_noun,Rang),Vars1,Vars2),
           append([(Rang,W,X)],Vars2,Vars));
             unify(Vars1,Vars))}.

abst_noun(W) --> [W],{abst_noun(W)}.

trans_verb(X,Y,P) --> [W],
{(infinitive(I,W),
  trans_verb(I),
  P=..[I,X,Y]);
```

```
    (trans_verb(W),
     P=..[W,X,Y])}.

intrans_verb(X,P) --> [W],
        {(infinitive(I,W),
          intrans_verb(I),
          P=..[I,X]);
            (intrans_verb(W),
             P=..[W,X])}.

adj(X,P) --> [W],{adj(W),P=..[W,X]}.

prep(Rep1,Rep2,P) --> [W],{prep(W),P=..[W,Rep1,Rep2]}.

mod_prep(Rep1,Rep2,P) --> [W],{mod_prep(W),P=..[W,Rep1,Rep2]}.

name(N) --> [N],{name(N)}.

number(N) --> [nb(N)],{number(N)}.

year(X) --> [nb(X)],{year(X)}.
```

para.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%% -*- Mode: Prolog -*- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% para.pl ---
%%% Author          : Anders Blehr
%%% Created On      : Tue Apr 21 13:53:37 1992
%%% Last Modified By: Anders Blehr
%%% Last Modified On: Sun May  3 11:30:22 1992
%%% RCS revision    : $Revision: 1.5 $ $Locker: blehr $
%%% Status          : Under development
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

paraphrase(Type,Rep) :-
        call_para(Type,Rep,Sent),
        write_sent(Sent).

call_para(rule,Rep1=>Rep2,Sent) :-
        call_para(rule,Rep2,Sent1),
        call_para(rule,Rep1,Sent2),
        append(Sent1,[if],TmpSent),
        append(TmpSent,Sent2,Sent),!.

call_para(Type,Rep,Sent) :-
        copy_term(Rep,Rep1),
        numbervars(Rep1,0,_),
        listify(Rep1,ListRep),
        call_sent(Type,Sent,ListRep),!.

call_sent(rule,Sent,List) :- sent(Sent,List,[]).
call_sent(noquery,Sent,List) :-
        sent(Sent1,List,[]),
        extract_vars(Sent1,Sent2,[],Vars,[]),
        replace_vars(Sent2,Sent,Vars).

call_sent(query,Sent,List) :-
        query_sent(Wh,Sent1,[],Vars,List,[]),
        replace_vars(Sent1,Sent2,Vars),
        compose_question(Wh,Sent2,Sent).

%%

query_sent(Wh,Sent,Vars1,Vars) --> [which,'(',X,')',':','(',Q,'(',X,')',&],
        query_sent(_,Sent,Vars1,Vars),[')'],{get_wh_pron(Q,Wh)},!.

query_sent(Wh,Sent,Vars1,Vars) --> [exists,'(',X,')',':','('],
        query_simple(Sent1),{append([(X,Sent1)],Vars1,Vars2)},[&],
        query_sent(Wh,Sent,Vars2,Vars),[')'],!.

query_sent(Wh,Sent,Vars1,Vars) -->
        prep(P),['('],
        query_sent(Wh,Sent1,Vars1,Vars),[','],
        atom_term(A),[')'],{append(Sent1,[P,A],Sent)},!.

query_sent(is,[S,is|Rest],Vars,Vars) --> simple([S,is|Rest]),!.
query_sent(does,[S,V|Rest],Vars,Vars) --> simple([S,V|Rest]),!.
```

```
sent(Sent) --> [exists,'(',_,')',':','('],
        sent(Sent1),[')',&],
        sent(Sent2),
        {append(Sent1,[and],TmpSent),
         append(TmpSent,Sent2,Sent)},!.

sent(Sent) --> [exists,'(',_,')',':','('],
        sent(Sent),[')'],!.

sent(Sent) -->
        simple(Sent1),[&],
        sent(Sent2),
        {append(Sent1,[and],TmpSent),
         append(TmpSent,Sent2,Sent)},!.

sent(Sent) --> simple(Sent),!.

query_simple([Art,Noun]) --> noun_simple([_,is,Art,Noun]),!.
query_simple([Art,Noun|Rest]) --> prep_simple([_,is,Art,Noun|Rest]),!.

simple(Sent) --> noun_simple(Sent),!.
simple(Sent) --> adj_simple(Sent),!.
simple(Sent) --> iverb_simple(Sent),!.
simple(Sent) --> tverb_simple(Sent),!.
simple(Sent) --> prep_simple(Sent),!.

prep_simple(Sent) -->
        prep(P),['('],
        sent(Sent1),[','],
        atom_term(A),[')'],{append(Sent1,[P,A],Sent)},!.

prep_simple([A1,is,P,A2]) -->
        prep(P),['('],
        atom_term(A1),[','],
        atom_term(A2),[')'],!.

adj_simple([A,is,Adj]) -->
        adj(Adj),['('],
        atom_term(A),[')'].

noun_simple([A,is,Art,N]) -->
        noun(N),['('],{get_article(N,Art)},
        atom_term(A),[')'].

iverb_simple([A,IV]) -->
        iverb(IV1),['('],{present(IV1,IV)},
        atom_term(A),[')'].

tverb_simple([A1,TV,A2]) -->
        tverb(TV1),['('],{present(TV1,TV)},
        atom_term(A1),[','],
        atom_term(A2),[')'].

prep(P) --> [P],{prep(P)}.
```

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

noun(N) --> [N],{noun(N)}.

iverb(IV) --> [IV],{intrans_verb(IV)}.

tverb(TV) --> [TV],{trans_verb(TV)}.

atom_term(A) --> [A],{atom_term(A)}.

%%

atom_term(A) :- name(A),!.
atom_term(A) :- abst_noun(A),!.
atom_term(A) :- time(A),!.
atom_term(A) :- var_term(A),!.

var_term(X) :- X=..['$VAR',_].

get_wh_pron(place,where).
get_wh_pron(W,who) :- member(W,[man,woman,person,child]),!.
get_wh_pron(W,when) :- member(W,[time,year]),!.

get_subject([Art,Noun],[Art,Noun|Sent],Sent) :- member(Art,[a,an]),!.
get_subject([S],[S|Sent],Sent) :- atom_term(S),!.

compose_question(does,Sent1,Sent) :-
        get_subject(Subj,Sent1,Sent2),
        pop(V1,Sent2,Sent3),
        infinitive(V,V1),
        append([does],Subj,Sent4),
        append(Sent4,[V],Sent5),
        append(Sent5,Sent3,Sent6),
        append(Sent6,['?'],Sent),!.

compose_question(is,Sent1,Sent) :-
        get_subject(Subj,Sent1,Sent2),
        pop(is,Sent2,Sent3),
        append([is],Subj,Sent4),
        append(Sent4,Sent3,Sent5),
        append(Sent5,['?'],Sent),!.

compose_question(who,Sent1,Sent) :-
        pop(S,Sent1,Sent2),
        var_term(S),
        append([who],Sent2,Sent3),
        append(Sent3,['?'],Sent),!.

compose_question(who,Sent1,Sent) :-
        get_subject(Subj,Sent1,Sent2),
        pop(V1,Sent2,Sent3),
        get_subject(_,Sent3,Sent4),
        infinitive(V,V1),
        append([whom,does],Subj,Sent5),
        append(Sent5,[V],Sent6),
        append(Sent6,Sent4,Sent7),
        append(Sent7,['?'],Sent),!.

compose_question(Wh_pron,[S,is,Adj|_],Sent) :-
        member(Wh_pron,[when,where]),
        unify(Sent,[Wh_pron,is,S,Adj,'?']),!.

compose_question(Wh_pron,Sent1,Sent) :-
        member(Wh_pron,[when,where]),
        get_subject(Subj,Sent1,Sent2),
        pop(V1,Sent2,Sent3),
        infinitive(V,V1),
        append([Wh_pron,does],Subj,Sent4),
        append(Sent4,[V],Sent5),
        append(Sent6,[in|_],Sent3),
        append(Sent5,Sent6,Sent7),
        append(Sent7,['?'],Sent),!.

replace_vars([],[],_).
replace_vars([VarS|Rest1],Sent,Vars) :-
        var_term(VarS),
        find_var(VarS,ListS,Vars),
        \+ (ListS=[nil|_]),
        replace_vars(Rest1,Rest,Vars),
        append(ListS,Rest,Sent),!.

replace_vars([W|Rest1],[W|Rest],Vars) :- replace_vars(Rest1,Rest,Vars),!.

extract_vars(Sent1,Sent,Vars1,Vars,Visited1) :-
        append([S,is,Art,N|SRest],[and|Rest],Sent1),
        var_term(S),
        \+ member(S,Visited1),
        member(Art,[a,an]),
        append([S],Visited1,Visited),
        append([(S,[Art,N|SRest])],Vars1,Vars2),
        extract_vars(Rest,Sent,Vars2,Vars,Visited),!.

extract_vars(Sent1,Sent,Vars1,Vars,Visited) :-
        append([S,is,Adj],[and|Rest],Sent1),
        member(S,Visited),
        adj(Adj),
        get_article(Adj,Art),
        pop((S,[_|SRest]),Vars1,Vars2),
        append([(S,[Art,Adj|SRest])],Vars2,Vars3),
        extract_vars(Rest,Sent,Vars3,Vars,Visited).

extract_vars(Sent1,Sent,Vars1,Vars,Visited) :-
        append(Sent2,[and|Rest],Sent1),
        extract_vars(Rest,Sent3,Vars1,Vars,Visited),
        append(Sent2,[and],TmpSent),
        append(TmpSent,Sent3,Sent),!.

extract_vars(Sent,Sent,Vars,Vars,_).

find_var(_,(nil,nil),[]).
find_var(VarS,S,Vars) :- pop((VarS,S),Vars,_),!.
find_var(VarS,S,Vars) :-
        pop(_,Vars,Vars1),
        find_var(VarS,S,Vars1),!.

write_sent([]).
write_sent(Sent) :-
        pop(W,Sent,Sent1),
        write(W),
        write(' '),
        write_sent(Sent1).
```

readin.pl

This file contains input and output predicates, which for the most part are implemented by Tore Amble. Some predicates have been slightly modified by me.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%% -*- Mode: Prolog -*- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% readin.pl ---
%%% Author          : Tore Amble
%%% Created On      : Sun May  5 1991
%%% Last Modified By: Anders Blehr
%%% Last Modified On: Mon Apr 27 10:09:19 1992
%%% RCS revision    : $Revision: 1.8 $ $Locker: blehr $
%%% Status          : Working
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

read_sentence(user,P) :- !,
        ttyflush,
        read_in(P).

read_sentence(_,P) :-
        read_in(P),
        doing(P,0),
        write(' */'),
        nl.

%%% Read a Sentence

read_in(P) :-
        initread(L),
        words(P,L,[]),!.

read_line(P) :-
        readline(L),
        words(P,L,[]). %%  TA-900606

initread([K1|U]) :-
        get(K1),
        readrest(K1,U).

readrest(A,[K1|U]) :-
        appostrophe(A),!,
        get0(K1),
        readrestquote(A,K1,U).

readrest(T,[]) :-
        term_char(T),!,  %% .?! read outside quote
        to_nl.

readrest(C,[]) :-
        comment_char(C),!,
        to_nl.

readrest(K,[K1|U]) :-
        K=<32,!,
        get(K1),
        readrest(K1,U).

readrest(_,[K2|U]) :-
        get0(K2),
        readrest(K2,U).

readrestquote(A,A,[K1|U]) :-!, %% found matching appostrophe
        get(K1),
        readrest(K1,U).

readrestquote(A,_,[K1|U]) :- !, %% still inside quotes
        get0(K1),
        readrestquote(A,K1,U).

readline([K|R]) :-
        get0(K),
        \+ K=10,!, % CR
        readline(R).

readline([]).

comment_char(37). %-)

term_char(33).
term_char(46).
term_char(63).

words([V|U]) -->
        wordc(V),!,
        blanks,
        words(U).

words([]) --> [].

wordc(W) --> word1(CL,Type),{w01(CL,Type,W)}.

w01(CL,V,nb(U)) :-
        var(V),!,
        name(U,CL). % atom_chars(U,CL).

w01(CL,quote,quote(U)) :- !,atom_chars(U,CL).
w01(CL,_,U) :- !,atom_chars(U,CL).

word01([X|Y],Type) -->
        idchar(X,Type),
        word01(Y,Type).

word01([],_) --> [].

idchar(X,alpha) --> [C],{lc(C,X)}.
idchar(X,alpha) --> [X],{sign(X)}.
idchar(X,_) --> [X],{digit(X)}.

word1([X|Y],Type) -->
        idchar(X,Type),
        word01(Y,Type).

word1(S,quote) -->
        [P],{appostrophe(P)},!,
        charstring(S),
        [P].

word1([K],term) --> [K],{not_appostrophe(K)}.

charstring([X|Y]) -->
        [X], {not_appostrophe(X)},
        charstring(Y).

charstring([]) --> [] .

appostrophe(96). %% `
appostrophe(39). %% '
appostrophe(34). %% "

not_appostrophe(X) :- \+ appostrophe(X).

%% TA

alphanums([K1|U]) -->
        [K],{alphanum(K,K1)},!,
        alphanums(U).

alphanums([]) --> [].

alphanum(K,K1):-lc(K,K1).
alphanum(K,K):-digit(K).
alphanum(K,K):-sign(K).

digits([K|U]) -->
        [K],{digsign(K)},!,
        digits(U).

digits([]) --> [].


digsign(X) :- digit(X).
digsign(X) :- sign(X).

digit(K) :-
        K>47,
        K<58.

sign(95) :- !. %% _
sign(45) :- !. %% -

blanks -->
        [K],{K=<32},!,
        blanks.

blanks --> [].
```

```
alpha(Y) --> [X],{lc(X,Y)}.

lc(91,123). %% AE
lc(92,124). %% OE
lc(93,125). %% AA


lc(K,K1) :-
        K>64,
        K<91,!,
        K1 is K\/8'40.

lc(K,K) :-
        K>96,
        K<126.

to_nl :-
        repeat,
        get0(10),!.

%%% Miscellaneous for File Reading

doing([],_) :- !.
doing([X|L],N0) :-
        out1(X),
        advance(X,N0,N),
        doing(L,N).

out1(nb(X)) :- !,write(X).

out1(A) :- write(A).

advance(X,N0,N) :-
        uses(X,K),
        M is N0+K,
        (M>72,!,
         nl,
         N is 0;
            N is M+1,
            put(" ")).

uses(nb(X),N) :- !,chars(X,N).
uses(X,N) :- chars(X,N).

chars(X,N) :- atomic(X),!,
        atom_chars(X,L),
        length(L,N).

chars(_,2).
```

remember.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%% -*- Mode: Prolog -*- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% remember.pl ---
%%% Author          : Anders Blehr
%%% Created On      : Wed Apr 15 19:24:58 1992
%%% Last Modified By: Anders Blehr
%%% Last Modified On: Sun May  3 11:30:53 1992
%%% RCS revision    : $Revision: 1.10 $ $Locker: blehr $
%%% Status          : Under development
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rem(_,before(_,_)).
rem(_,forall(_):(R1=>R2)) :- remember(rule(R1=>R2)).
rem(_,R1=>R2) :- remember(rule(R1=>R2)).
rem(subject(S),R) :- explode_it(subject(S),known(R)).

remember(X) :-
        copy_term(X,X1),
        numbervars(X1,0,_),
        call(X1),!.

remember(X) :- assert(X).

explode_it(_,known(X)) :- known(X),!.
explode_it(subject(S),known(X)) :- unnest_it(subject(S),X).

unnest_it(subject(S),exists(X):(RepS&Rep1)) :-
        var(S),
        S==X,
        remove_instQs(Rep1,Rep),
        unnest([existensial(X),RepS],Rep,[],_).

unnest_it(subject(S),Rep1) :-
        remove_instQs(Rep1,Rep),
        unnest([S],Rep,[],_).

unnest(_,X,_,_) :- var(X).
unnest(Subject,Rep1&Rep2,_,_) :-
        unnest(Subject,Rep1,[],_),
        unnest(Subject,Rep2,[],_).

unnest(Subject,exists(X):(Rep1&Rep2),Pairs1,Pairs) :-
        Rep2=..[P,Rep3,Rep4],
        append([existensial(X),Rep1,(P,Rep4)],Pairs1,Pairs2),
        assert_all(Subject,Rep3,Pairs2),
        unnest(Subject,Rep3,Pairs2,Pairs).

unnest(Subject,Rep,Pairs1,Pairs) :-
        Rep=..[P,Rep1,Rep2],
        append([(P,Rep2)],Pairs1,Pairs2),
        assert_all(Subject,Rep1,Pairs2),
        unnest(Subject,Rep1,Pairs2,Pairs).

unnest(Subject,Rep1,Pairs,Pairs) :-
        Rep1=..[_,_],
        append_varsubj(Subject,Rep1,Rep),
        remember(known(Rep)).

unnest(_,_,_,_).

assert_all(_,_,[]).
assert_all(Subject,Rep1,Pairs) :-
        get_prefix(Pairs,Pairs1,Prefix),
        pop((P,Q),Pairs1,Pairs2),
        Rep2=..[P,Rep1,Q],
        make_term(Prefix,Rep2,Rep3),
        append_varsubj(Subject,Rep3,Rep),
        remember(known(Rep)),
        assert_tail(Prefix,Q,Pairs2),
        assert_all(Subject,Rep1,Pairs2).

assert_tail(_,_,[]).
assert_tail(Prefix1,Rep1,Pairs) :-
        get_prefix(Pairs,Pairs1,Prefix2),
        pop((P,Q),Pairs1,Pairs2),
        Rep2=..[P,Rep1,Q],
        make_term(Prefix1,Rep2,Rep3),
        make_term(Prefix2,Rep3,Rep),
        remember(known(Rep)),
        assert_tail(Prefix1,Rep1,Pairs2).

get_prefix(Pairs1,Pairs,exists(X):Rep2) :-
        pop(existensial(X),Pairs1,Pairs2),
        pop(Rep2,Pairs2,Pairs).

get_prefix(Pairs,Pairs,noprefix).

make_term(noprefix,Rep,Rep).
make_term(exists(X):Rep1,Rep2,exists(X):(Rep1&Rep2)).

append_varsubj([existensial(X),RepS],Rep,exists(X):(RepS&Rep)).
append_varsubj([_],Rep,Rep).

remove_instQs(exists(X):(_&Rep1),Rep) :-
        nonvar(X),
        remove_instQs(Rep1,Rep).

remove_instQs(exists(X):(RepQ&Rep1),exists(X):(RepQ&Rep)) :-
        remove_instQs(Rep1,Rep).

remove_instQs(Rep1&Rep2,Rep3&Rep4) :-
        remove_instQs(Rep1,Rep3),
        remove_instQs(Rep2,Rep4).

remove_instQs(Rep1,Rep) :-
        Rep1=..[P,Rep2,R],
        nonvar(Rep2),
        remove_instQs(Rep2,Rep3),
        Rep=..[P,Rep3,R].
```

```
remove_instQs(Rep,Rep).
```

solve.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%% -*- Mode: Prolog -*- %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% solve.pl ---
%%% Author          : Anders Blehr
%%% Created On      : Sun Apr 12 15:19:00 1992
%%% Last Modified By: Anders Blehr
%%% Last Modified On: Sun May  3 11:31:03 1992
%%% RCS revision    : $Revision: 1.11 $ $Locker: blehr $
%%% Status          : Under development
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

which_solve(which(X):(RepQ&Rep),Quant1,Quant) :-
known(Rep),
confirm_quantifier(exists(X):RepQ,Quant1,Quant).

solve(Q,Quant,Quant,Proof1,Proof) :-
known(Q),
append_proof(Q,Proof1,Proof),!.

solve(exists(X):(_&Rep),Quant1,Quant,Proof1,Proof) :-
nonvar(X),
solve(Rep,Quant1,Quant,Proof1,Proof),
rem(_,Rep).

solve(exists(X):(Q&Q1),Quant1,Quant,Proof1,Proof) :-
solve(Q1,Quant1,Quant2,Proof1,Proof),
confirm_quantifier(exists(X):Q,Quant2,Quant),
rem(_,Q1).

solve(after(X,Y),Quant,Quant,Proof1,Proof) :-
\+ solve(before(Y,X),_,_),
append_proof(after(X,Y),Proof1,Proof),!.

solve(before(X,Y),Quant,Quant,Proof1,Proof) :-
number(X),
number(Y),
X=<Y,
append_proof(before(X,Y),Proof1,Proof),!.

solve(Q1&Q2,Quant,Quant,Proof1,Proof) :-
breakup(Q1&Q2,[],Qs),
\+ (Qs=[]),
loop_solve(Qs,Quant,Proof1,Proof).

solve(Q,Quant1,Quant,Proof1,Proof) :-
rule(Q1=>Q),
remove_instQs(Q1,Q2),
copy_term(Q2,Q3),
get_quantifiers(Q2,Quant1,Quant2),
solve(Q2,Quant2,Quant,Proof1,Proof2),
numbervars(Q3,0,_),
append_proof(Q3=>Q,Proof2,Proof),!.

solve(Q,Quant1,Quant,_,_) :-
subconclusion(Q,Q1),
remove_instQs(Q1,Q2),
solve(Q2,Quant1,Quant,[],_),!.

solve(Q,Quant,Quant,Proof1,Proof) :-
Q=..[W,_],
member(W,
[city,country,dead,live,man,person,place,time,woman,year]),
call(Q),
append_proof(Q,Proof1,Proof),!.

solve(Q,Quant1,Quant,Proof1,Proof) :-
\+ Q=exists(_):_,
Q=..[P,Q1,R],
nonvar(Q1),
\+ num_of_vars(Q1,0),
solve(Q1,Quant1,Quant2,Proof1,Proof2),
remove_instQs(Q1,Q2),
Q3=..[P,Q2,R],
solve(Q3,Quant2,Quant,Proof2,Proof3),
rem(_,Q3),
append_proof(Q3,Proof3,Proof),!.

solve(Q,Quant1,Quant,Proof1,Proof) :-
\+ Q=before(_,_),
\+ Q=exists(_):(_&before(_,_)),
inquire(Q,Quant1,Quant),
remember_it(Q),
append_proof(Q,Proof1,Proof),!.

loop_solve([],_,Proof,Proof) :- !.
loop_solve(Qs,Quant,Proof1,Proof) :-
pop(Q,Qs,RQs),
solve(Q,Quant,Quant1,Proof1,Proof2),
loop_solve(RQs,Quant1,Proof2,Proof),!.

loop_solve(Qs,Quant,Proof1,Proof) :-
pop(Q,Qs,TmpQs),
append(TmpQs,[Q],NQs),
loop_solve(NQs,Quant,Proof1,Proof),!.

get_quantifiers(Rep1&Rep2,Quant1,Quant) :-
get_quantifiers(Rep1,Quant1,Quant2),
get_quantifiers(Rep2,Quant2,Quant).

get_quantifiers(exists(X):(RepQ&Rep),Quant1,Quant) :-
var(X),
append([(X,RepQ)],Quant1,Quant2),
get_quantifiers(RepQ&Rep,Quant2,Quant).

get_quantifiers(Rep,Quant1,Quant) :-
Rep=..[_,Rep1,_],
nonvar(Rep1),
get_quantifiers(Rep1,Quant1,Quant).

get_quantifiers(_,Quant,Quant).
```

```
confirm_quantifier(exists(X):(Q1&Q2),Quant1,Quant) :-
solve(Q2,Quant1,Quant2,[],_),
confirm_quantifier(exists(X):Q1,Quant2,Quant).

confirm_quantifier(exists(_):Q,Quant1,Quant) :- solve(Q,Quant1,Quant,[],_).

breakup(Q1&Q2,Qs,NQs) :-
append([Q1],Qs,TmpQs),
breakup(Q2,TmpQs,NQs).

breakup(Q,Qs,NQs) :- append([Q],Qs,NQs).

subconclusion(Q,Q1) :- rule(Q1=>Q&_).
subconclusion(Q,Q1) :- rule(Q1=>_&Q).
subconclusion(Q,Q1) :- rule(Q1=>_&Q&_).

get_var(_,_,[],[]) :- !,fail.
get_var(X,Rep,Quant1,Quant) :-
pop((Y,Rep),Quant1,Quant),
Y==X.

get_var(X,Rep,Quant1,Quant) :-
pop(Tmp,Quant1,Quant2),
get_var(X,Rep,Quant2,Quant3),
append([Tmp],Quant3,Quant).

num_of_vars(Q,N) :-
copy_term(Q,Q1),
numbervars(Q1,0,X),
X=N.

remember_it(Q) :-
num_of_vars(Q,0),
rem(_,Q).

remember_it(_).

%%

inquire(exists(X):Q,Quant,Quant) :-
var(X),
num_of_vars(exists(X):Q,1),
ask(exists(X):Q,Quant,_),!.

inquire(Q,Quant1,Quant) :-
num_of_vars(Q,1),
append_quantifier(Q,Q1,Quant1,Quant2),
ask(Q1,Quant2,Quant),!.

inquire(Q,Quant,Quant) :-
num_of_vars(Q,0),
ask(Q,_,_),!.

ask(Rep,_,_) :- known(Rep),!.
ask(exists(X):(RepQ&Rep),Quant1,Quant) :-
copy_term(which(X):(RepQ&Rep),Q),
numbervars(Q,0,_),
get_input(Q,L),
((L=[nb(Reply)|_];L=[Reply|_]),
  \+ Reply=no,
  unify(X,Reply),
  confirm_quantifier(exists(X):RepQ,Quant1,Quant));
L=proceed,
solve(exists(X):(RepQ&Rep),Quant1,Quant,[],_),!.

ask(Q,Quant,Quant) :-
num_of_vars(Q,0),
get_input(Q,L),
L=[yes|_],!.

append_quantifier(exists(X):(_&Rep),Rep,Quant,Quant) :- nonvar(X),!.
append_quantifier(Q,AskableQ,Quant1,Quant) :-
((Q=..[_,X,_],var(X));(Q=..[_,_,X],var(X))),
get_var(X,Rep,Quant1,Quant),
unify(AskableQ,exists(X):(Rep&Q)),!.

append_quantifier(Q,exists(X):(Rep&Q),Quant1,Quant) :-
Q=..[_,Rep1,_],
nonvar(Rep1),
Rep1=..[_,X],
var(X),
get_var(X,Rep,Quant1,Quant),!.

append_proof(Q,Proof1,Proof) :-
copy_term(Q,Q1),
numbervars(Q1,0,_),
append([Q1],Proof1,Proof).

get_input(Q,L) :-
paraphrase(query,Q),
write('  '),
read_sentence(user,L1),
evaluate_input(L1,L).

evaluate_input(L1,proceed) :-
sentence(subject(X),Rep,Type,L1,[]),
evaluate(Type,subject(X),Rep),!.

evaluate_input(L,L).
```

Bibliography

[1] Tore Amble. Understanding systems with second order logic. Technical report.

[2] Tore Amble. *Logic Programming and Knowledge Engineering.* Addison-Wesley Publishing Company, 1987.

[3] Tore Amble. Representasjon av regelverk i naturlig lesbar logikk - norsk lov om statsborgerrett som ekspertsystem. Lecture notes, 1987.

[4] Mats Carlsson. *SICStus Prolog User's Manual.* Swedish Institute of Computer Science, PO Box 1263, S-16428 KISTA, Sweden. Draft version: 21 February 1992.

[5] Jorunn Eggen and Bård Hugaas. Ekspertsystemskall i Prolog. Master's thesis, The Norwegian Institute of Technology (NTH), 1984.

[6] Annie Gal, Guy Lapalme, Patrick Saint-Dizier and Harold Somers. *Prolog for Natural Language Processing.* John Wiley & Sons, 1991.

[7] H. P. Grice. Logic and conversation. In P. Cole and J. L. Morgan, editors, *Syntax and Semantics*, volume 3 of *Speech Acts*, pages 41–58. Academic Press, New York, 1975.

[8] A. G. Hamilton. *Logic for Mathematicians.* Cambridge University Press, 1988.

[9] Leslie Lamport. *A Document Preparation System - LATEXUser's Guide & Reference Manual.*

[10] Åsmund Mæhle. Ekspertsystem for loven om norsk statsborgerskap med brukergrensesnitt i naturlig språk. Project Report, NTH, 1990.

[11] Torulf Mollestad. Natural language interface to databases: Cooperating with the casual user. Master's thesis, The Norwegian Institute of Technology (NTH), 1989.

[12] Ulf Nilsson and Jan Małuszyński. *Logic, Programming and Prolog.* John Wiley & Sons, 1990.

[13] Fernando C. N. Pereira and Stuart M. Shieber. *Prolog and Natural-Language Analysis.* Center for the Study of Language and Information, 1987. Lecture notes.

[14] Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

[15] Elaine Rich and Kevin Knight. *Artificial Intelligence.* McGraw-Hill, Inc., 1991.

[16] M. J. Sergot, F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond and H. T. Cory. The British nationality act as a logic program. *Communications of the ACM*, 29(5), May 1986.