Developing a Natural Language Interface and Connecting it to a First Order Logic Theorem Prover

by

Anna Bång and Per Lindberger
Developing a Natural Language Interface and Connecting it to a First Order Logic Theorem Prover

A Master of Science Thesis Work
by Anna Bång and Per Lindberger
Royal Institute of Technology, Stockholm
School of Computer Science and Engineering
September 1989

Funded by and performed at the
Swedish Institute of Computer Science (SICS)
Box 1263
S-164 28 Kista
SWEDEN
ABSTRACT

In this thesis we describe and develop a simple natural language interface for AI applications. The interface is based on TALK, a system originating from Fernando C. N. Pereira and Stuart M. Shieber. After a brief presentation of their system we delineate implementations of several indispensable language constructions. A complete example, a puzzle solving program combining the enhanced NL interface with a theorem prover, is included. Finally, we discuss and carry out a "purification" of our system, thus enabling it to run in a parallel logic programming environment not fully compatible with Prolog.

ACKNOWLEDGEMENTS

We would like to thank a number of researchers and administrative staff at the Swedish Institute of Computer Science for help and guidance, especially Manny Rayner, who took a lot of time away from his work to assist and inspire us in completing this thesis.
5 A Pure TALK .................................................................31
  5.1 The expansion of DCG rules to Prolog clauses ..................31
  5.2 Avoiding calls to extralogical predicates in TALK .............32
  5.3 Replacing Prolog variables in the logical forms ................33

6 Conclusion .......................................................................35

Appendix A The extended NL system .....................................36

Appendix B The puzzle solving program .................................46
  B.1 The top level program ...............................................46
  B.2 The natural language processing part .........................48

References ...........................................................................57
1 Introduction

1.1 Natural language and computers

Before the invention of the digital computer, language was seen as the exclusive province of human beings. The signaling systems of other animals were too rigid and simple to deserve the designation language, and all that mechanical and electrical devices could do was to store and transmit sequences of codes to be interpreted by people. With the computer, new possibilities appeared. The essential quality of the digital computer being its ability to manipulate symbols (not just numbers, but symbols of practically any kind), it was recognized from the very earliest days of computing that, in addition to the obvious applications in scientific calculation and book keeping, computers could work with language.

Since computer technology included the development of specialized artificial languages such as programming languages, computer scientists adopted the practice of logicians and mathematicians (who also develop formal languages) to use the term natural language for language spoken by humans. A wide range of potential applications of computers to language has been proposed over the years and the following are just a few examples from a long list of topics being explored.¹

Machine translation is the automatic translation of texts (particularly technical and scientific papers) from one language to another. Although this was one of the first areas to be investigated, it seems unlikely that a fully successful system will ever be made. However, it has been argued that machine translation can considerably reduce the cost of translation and every year millions of dollars are spent on its development.

Information retrieval is the name of another active area in which research is being pursued. Computers have the capacity of storing arbitrarily large texts and it would be useful to be able to ask and receive answers to questions concerning these texts in a natural language.

Speech recognition and speech synthesis are vital in systems trying to communicate in spoken language so as to have a more flexible, relaxed way of interaction between humans and computers. Though research in these areas has barely begun, there has been some results.

Knowledge acquisition. Some complex systems (often referred to as expert systems) are based on a large body of stored knowledge about a particular problem area. In the building of such systems the programmer must incorporate bodies of material that are known to experts in the field, a task

¹ For further information on these points and many others see [17].
which requires interaction with those experts and can be both tedious and costly.

Some of the current work on expert systems consists of efforts to provide some form of natural language interaction between the program and an expert, through which the program's body of knowledge can be built. Some longer term research based on the hope that after reaching a certain level such systems could be given material in the form of text and incorporate the contents without human intervention is underway.

These and other examples justify the view held by many experts that natural language processing is one of the most important and challenging areas in today's computer science and artificial intelligence research.

1.2 A brief history of natural language processing

The history of Natural Language Processing (NLP) is generally considered to have begun in the early sixties with the building of several AI-systems involving NLP in a very primitive sense. The work on machine translation, mentioned in the previous section, focused a lot of interest on grammars and parsing (i.e., syntax). Research in this area has progressed continually for a long time, whereas in the field of semantics, with all its philosophical ramifications, progress seems slower and a lot remains to be done, notably in the domain of discourse.

Broadly speaking, research in computational semantics focused first on the sentence, alone or in its immediate context, and on determining a sentence's literal meaning. The latter relied heavily on systems being limited to specific domains and relatively straightforward tasks. The results of said research are now being developed and applied in commercial systems, e.g., for querying databases.

The following points attempt to describe the state of the art of NLP.

* A lot has been learned about parsing and syntax.

* Some limited semantic techniques exist, although efficient and comprehensive general approaches are still lacking.

* Work has begun on discourse interpretation as well as on language actions and intentions. Here also, much more needs to be done, e.g., on how to combine the ways of handling these with other system processes.

* The parts of natural-language-understanding research ready for development are primarily able to deal with literal meaning and direct function in a limited and well-defined task and domain context.

In conclusion: Little progress has been made in relation to what was wanted, since the field has proven more complex than expected.
Nevertheless, research proceeds in many directions and reports of imminent progress are emerging. For a list of references see the introduction of [7].

1.3 Grammars for natural language processing

Before making any attempt to process a language, the language in question must have been rigorously defined. This can be done through the use of a grammar, i.e. a collection of rules precisely stating which sequences of words and symbols comprise the sentences of the language and thereby implicitly excluding all other sequences. In addition, a useful NLP grammar generally gives some kind of analysis of sentences into structures or logical forms, making their meaning more explicit. In NLP research several types of grammar have been exploited and we attempt here to describe and compare but a few of the most popular ones: Augmented Transition Networks, Categorial Grammars and Definite Clause Grammars.

1.3.1 Augmented Transition Networks

One of the early models for NLP grammars was the finite state transition graph. Stemming from this, Augmented Transition Networks (ATNs) ([2], [16], [18]) began to appear in the early 1960’s. An ATN is basically equivalent to a NDPDA (Non Deterministic Push Down Automaton) and accordingly has nearly the computing power of a Turing Machine. Research on ATNs began when it became apparent that Context Free Grammars (CFGs) aren’t quite as suitable for NLP as for defining formal languages. For instance, many NL sentences require some degree of agreement between distant parts of each sentence; CFGs are inadequate in this respect. Among the major features that an ATN adds to those of a CFG we find the ability to store information about parsed subparts of a sentence for consistency checking and the capacity to move, copy and delete fragments of a sentence. This enables the ATN to generate almost any syntactic structure for a given sentence.

Below is an example of an (unaugmented) Transition Network, that together with a suitable lexicon accepts sentences like: John loves a woman and Mary sleeps.³

---

1 Conway’s paper, [5], provides one of the earliest descriptions of ATNs.
2 There are those who hold a different view though. Gazdar et al. argue that NL can be regarded as CFGs.
3 For an example of how an augmented version can be implemented in LISP refer to [18].
The traversal of an arc with a state name as its label causes the pushing of the state at the end of the arc onto a stack and control jumps to the state named on the arc. Encountering a final state (the circles drawn in a thicker style) the automaton pops the top-most state of the stack and transfers control to it. Reaching a final state with nothing left to pop means that the parsing was successful and the input string accepted.

1.3.2 Categorial Grammars

Categorial grammars have been widely used in linguistic research concerning semantics of natural language (cf [12]). A Categorial Grammar (CG) lacks phrase structure rules; instead it describes the possibilities of combining lexical items directly. The grammar divides phrases into categories that encode the ways in which they can combine with other phrases. An inductive definition of a set of categories might look like:

* A primitive category is a category (primitive categories may be VerbPhrases (VPs) or NounPhrases (NPs), for instance).

* If A and B are categories then A/B and A\B are categories, e.g. S\NP is a category if S and NP are categories.

* Nothing else is a category.

The legitimate ways of combining phrases are defined by the following rules:

* Forward application: A phrase p of category A/B can be combined with a phrase q of category B to form a phrase pq of category A.

* Backward application: A phrase p of category A\B can be combined with a phrase q of category B to form a phrase qp of category A.

To illustrate a CG, we give here the lexical part of a CG implementation of the grammar in figure 1. The primitive categories

---

1 The very influential linguist Richard Montague has done some important work on CGs, as have others.
needed are S, NP and N, denoting Sentences, Noun Phrases and Nouns respectively.

\begin{verbatim}
lex(john, NP).
lex(mary, NP).
lex(sleeps, S\NP).
lex(loves, (S\NP)/NP).
lex(a, NP/N).
lex(woman, N).
\end{verbatim}

...  

With the advent of logic programming, ATNs, CGs\(^1\) and other older grammars subsided to make way for a newer formalism: Definite Clause Grammars (DCGs), whose popularity hasn't been diminished by the fact that they are directly executable in Prolog, by far the most popular logic programming language ([14],[15]). Such grammars are the subject of the next subsection.

1.3.3 Definite Clause Grammars

Definite Clause Grammars extend Context Free Grammars, CFGs\(^2\), which have been widely used in computer science in the notation of BNF (Backus-Naur Form). The idea to translate CFGs into a more general formalism, viz. First Order predicate Logic (FOL), originated from Colmerauer and Kowalski when they devised a method for expressing context-free rules as logic statements of a restricted kind, known as Definite Clauses or Horn Clauses\(^3\), [10],[4]. The problem of recognizing, or parsing, a given sentence can then be reduced to the problem of proving a given theorem.

Here is how CFGs are generalized into the formalism of DCGs:

(1) Non-terminals can be arbitrary terms, not only atoms as in CFGs, e. g. np(NP) and vp(VP) are non-terminals.

(2) The right-hand side of a rule may contain non-terminals, lists of terminals and, within braces, sequences of procedure calls.

A DCG rule is nothing but a beautified way of writing a definite clause in Prolog: terminal symbols of arity N are the same as Prolog predicates of arity N + 2, the extra two arguments representing the list of words being parsed.\(^4\) Continuing with the same example we have:

---

\(^1\) Though recently there has been an interest in logic programming implementations of CGs.
\(^2\) The reader who wants to learn more about CFGs may refer to [1].
\(^3\) Horn Clauses or Definite Clauses are implications containing exactly one positive literal.
\(^4\) A more thorough discussion of this correspondence can be found in chapter 5.
\[ s \rightarrow np, vp. \]

\[ np \rightarrow pn. \]
\[ np \rightarrow det, n. \]

\[ vp \rightarrow iv. \]
\[ vp \rightarrow tv, np. \]

\[ pn(john). \]
\[ pn(mary). \]
\[ det(a). \]
\[ n(woman). \]

\[ iv(sleeps). \]
\[ tv(loves). \]

Figure 2. A simple DCG (left) and a sample lexicon.

None of the three versions of the grammar given here generates any structure, they merely recognize certain sentence constructions\(^1\).

1.3.4 A comparison between ATNs and DCGs

It should be noted at the outset that a complete and therefore just comparison between the two grammar formalisms is beyond the scope of a paper like the present. However, we feel that a restricted and perhaps subjective comparison could be of interest, if nothing else to emphasize why we have used DCGs and why they seem to be increasingly favoured over ATNs in NL literature and research.

One of the key advantages of DCGs over ATNs is their property of not only describing a machine for parsing a language but also being an obvious, declarative definition of the language under consideration. ATNs, on the other hand, are better described as automatons for parsing a particular language, and to understand how any but the very simplest ATN defines a language it seems inevitable to trace its execution. Furthermore, ATNs mingle two different representations - a graphical one, in the form of a transition network, and one which encodes the former in some appropriate language, e.g. LISP; a mixture absent in DCGs, where the grammar and its encoding are the same.

The degree to which a formalism can be understood and appreciated by researchers representing other fields is not irrelevant to its future use and development. In this respect DCGs go a long way towards providing a bridge between computer scientists involved in the design of NL systems on the one hand and linguists and philosophers on the other, whereas ATNs are more esoteric, thus not being able to draw upon the many important results in disciplines other than computer science.

Finally, as already mentioned, DCGs run in Prolog, a language that has advanced Artificial Intelligence (AI) and NLP research more than negligibly in recent years. This, in our opinion, makes DCGs the natural choice of grammar for NLP today.

---

\(^1\) We introduce the notion of structure generation in the next chapter.
2 TALK - A Simple NL System

TALK is the program chosen as a basis for the NL system we have developed. Here we include a brief description of TALK and at the same time introduce some notation needed for the sequel. The reader accustomed to DCGs, NL and Prolog can easily proceed to chapter 3, using this chapter as a reference.

The original version of TALK appears in an appendix of Pereira and Shieber: Prolog and Natural-Language Analysis [12]. Their intention was to illustrate how some simple NL implementations, discussed elsewhere in the book, could be put together. The program has a rather restricted utility since it had to be kept very small and clean, for educational purposes. As Pereira and Shieber point out, any attempt to extend TALK has to be subject of great care, because an extended version runs the risk of becoming muddled and incomprehensive, which is probably why they held on to their simplistic version. We have tried (chapter 3) to improve TALK by adding some new, seemingly essential constructions, without falling prey to excessive untidiness.

2.1 The grammar of TALK

The part of TALK that concerns itself with declarative sentences is a DCG with the following appearance.

%%%% Declarative sentences

\[
\text{s(\text{GapInfo})} \rightarrow \\
\text{np(\text{nogap}),} \\
\text{vp(\text{finite,GapInfo}).}
\]

- A sentence (s) is a noun phrase (np) followed by a verb phrase (vp).

%%%% Noun phrases

\[
\text{np(\text{nogap})} \rightarrow \\
\text{\text{det,}} \\
\text{\text{n,}} \\
\text{\text{optrel.}} \\
\text{np(\text{nogap})} \rightarrow \\
\text{\text{pn.}} \\
\text{np(\text{gap(np,X))} \rightarrow} \\
\text{\text{[]} .}
\]

- A noun phrase is either a determiner (det) followed by a noun (n) and an optional relative clause or a proper noun.

- This will be explained later.
Verb phrases

\[ \text{vp}(\text{Form}, \text{GapInfo}) \rightarrow \]
\[ \text{tv}(\text{Form}), \]
\[ \text{np}(\text{GapInfo}). \]
\[ \text{vp}(\text{Form}, \text{no-gap}) \rightarrow \]
\[ \text{iv}(\text{Form}). \]
\[ \text{vp}(\text{Form}, \text{GapInfo}) \rightarrow \]
\[ \text{aux}(\text{Form/Required}), \]
\[ \text{vp}(\text{Required}, \text{GapInfo}). \]
\[ \text{vp}(\text{finite}, \text{GapInfo}) \rightarrow \]
\[ [\text{is}], \]
\[ \text{np}(\text{GapInfo}). \]

Relative clauses

\[ \text{optrel} \rightarrow \]
\[ \text{relpron}, \]
\[ \text{vp}(\text{finite}, \text{no-gap}). \]
\[ \text{optrel} \rightarrow \]
\[ \text{relpron}, \]
\[ s(\text{gap}(\text{np}, X)). \]
\[ \text{optrel} \rightarrow \]
\[ [\text{is}]. \]

- A verb phrase is a transitive verb (tv) followed by a noun phrase or an intransitive verb (iv) or an auxiliary verb (aux) followed by another verb phrase or a copula followed by a noun phrase.

- A relative clause is a relative pronoun followed by a verb phrase or a relative pronoun followed by a sentence without a noun phrase or empty.

Many of the non-terminals in the DCG above take one or two arguments: \text{GapInfo} and/or \text{Form}. The need for and use of these are explained in the following subsections.

2.1.1 Information about gaps in sentences

Most sentences display an immediate adjacency between the subject and the predicate. However, in certain sentences with relative clauses, anaphora are allowed. In the DCG above, this problem is taken care of by an argument \text{GapInfo}, informing about gaps. \text{GapInfo} is added to certain rules and may be instantiated to \text{no-gap} or \text{gap(np,X)}, where the latter value indicates that an empty, implicit, noun phrase is allowed.

A relative clause consists of a relative pronoun and either a complete verb phrase or a sentence with a gap, and \text{GapInfo} will be instantiated accordingly (see the optrel rules above). This simple mechanism enables the DCG to allow for sentences like:

\[ \text{Every student ran a program that the professor wrote.} \]
\[ \text{Ivan who is a professor wrote a book.} \]

The first sentence exemplifies a case that invokes the optrel rule containing a gapped sentence. The second sentence is an instance of the other case: a regular verb phrase.

For a more thorough discussion of gaps in sentences see [12].
2.1.2 Information about verb forms in sentences

The grammar shown above doesn't support the concept of grammatical number at all, it does however incorporate a means of checking verb forms for correctness. For instance, in sentences containing auxiliary verbs there exists a relation between the verb form of the auxiliary verb and that of the verb phrase to follow, e.g., the auxiliary verb *be* requires its verb phrase to be of present participle form while the auxiliary verb *have* requires past participle form. The form of the auxiliary determines the form of the entire verb phrase.

The parameter *Form* deals with this particular problem. *Form* can have the values: *finite, nonfinite, present participle* and *past participle*. In the vp-rule that matches verb phrases beginning with an auxiliary verb it can be seen that aux has an argument of the form *Form/Required* where *Required* is determined by the auxiliary. Two sentences exemplifying these particular constructions follow:

*Ivan doesn't drink wine.*
*Ivan did phone Cleopatra.*

These sentences are accepted by TALK, as the verb phrases have the nonfinite form enforced by the auxiliary *do*, while the sentence

"Ivan did phones Cleopatra"

is rejected.

2.2 The dictionary of TALK, accompanied by a sample lexicon

A program like TALK presupposes the existence of a dictionary and a vocabulary, as do all useful NL systems. The dictionary of TALK is included here for completeness along with an arbitrary vocabulary. The dictionary entries all look like

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

where *lex* is the type of item to be matched and the singleton list [L] is a terminal symbol, i.e., a word of the sentence being parsed. The Prolog goal within braces, `lex(L)`, has to be true with respect to the vocabulary for the rule in question to succeed.

---

1 The asterisk serves to point out the ungrammaticality of the sentence, a convention adopted from the linguistics literature.
%% The dictionary

det -> [D], \{det(D)\}.
n -> [N], \{n(N)\}.
pn -> [PN], \{pn(PN)\}.

aux(Form) -> [Aux], \{aux(Aux,Form)\}.
relpron -> [RP], \{relpron(RP)\}.

iv(nonfinite) -> [IV], \{iv(IV,____)\}.
iv(finite) -> [IV], \{iv(____,IV,____)\}.
iv(finite) -> [IV], \{iv(____,IV,____)\}.
iv(past_participle) -> [IV], \{iv(____,IV,____)\}.
iv(pres_participle) -> [IV], \{iv(____,IV,____)\}.

tv(nonfinite) -> [TV], \{tv(TV,____)\}.
tv(finite) -> [TV], \{tv(____,TV,____)\}.
tv(finite) -> [TV], \{tv(____,TV,____)\}.
tv(past_participle) -> [TV], \{tv(____,TV,____)\}.
tv(pres_participle) -> [TV], \{tv(____,TV,____)\}.

%% The lexical items (the vocabulary)

det(every).
det(a).
det(an).
det(the).

n(man).
n(woman).
n(book).
n(wine).

pn(ivan).
pn(nero).
pn(cleopatra).

aux(does,finite/nonfinite).
aux(did,finite/nonfinite).

relpron(that).
relpron(who).

tv(love,loves,loved,loved,loving).
tv(write,writes,wrote,written,writing).
tv(phone,phones,phoned,phoned,phoning).
tv(drink,drinks,drank,drunk,drinking).

iv(run,runs,rain,run,running).
2.3 The meanings of sentences: introducing a logical form

Not only is TALK useful for recognizing syntactically correct sentences, it can also be of assistance in formalizing its semantics. This is done by the incorporation into the grammar of logical forms of sentences and constituents of sentences. Until now, we have deliberately omitted them from all the rules in TALK, so as to avoid unnecessary confusion. (Pereira and Shieber never present a grammar without logical forms, as we have done in the preceding section.)

In this section we introduce some notation and ideas. They are vital for the understanding of the rest of this report and should there be any obscurities, it may be wise to consult for instance [6].

2.3.1 The notation of lambda calculus in TALK

The entire production of logical forms for sentences in TALK is based on the well-known lambda calculus, introduced by Church in 1940 [3]. The notation adopted here differs slightly from the one commonly used. Instead of the usual

$$\lambda x. p(x)$$

we write

$$X^p(x).$$

Thus,

$$X^p X^p Z^p (X, q(Y, Z))$$

is equivalent to

$$\lambda x. \lambda y. \lambda z. p(x, q(y, z))).$$

2.3.2 Lambda reduction in TALK

Lambda reduction, i. e. the application of a lambda expression to an argument, is an important feature of the logical form synthesis in TALK. An example of traditional lambda reduction follows.

$$\lambda x. p(x) (a) \text{ reduces to } p(a).$$

The same example in TALK-notation looks like this:

reduce($X^p (X), a, \text{Reduced}$) will instantiate Reduced to p(a).
Reduce is a Prolog predicate defined by the simple fact

\[ \text{reduce(Args}^\text{Expr}, \text{Args}, \text{Expr}). \]

This very simple approach is sufficient in most cases, but we will see later (chapter 3), that more cumbersome techniques must be used on occasion.

In Pereira's and Shieber's original TALK, partial execution takes care of what would normally be referred to a call to \text{reduce}. Here is how a rule with a call to \text{reduce} can be rewritten by partially executing it:

\[ s(S) \rightarrow \text{np(NP)}, \text{vp(VP)}, [\text{reduce(VP,NP,S)}]. \]

is equivalent to

\[ s(S) \rightarrow \text{np(NP)}, \text{vp(NP}^\text{S}). \]

As the example indicates, partial execution of calls to \text{reduce} renders more compact, albeit not more legible, code.

2.4 TALK with logical forms

In the version of TALK presented in this section, each grammatical rule has been given an extra argument, to make possible the successive building of the logical forms. The forms are constructed concurrently with the parsing of sentences, as each rule has its own way of influencing the final result. In the process of putting together the form, extensive use is made of lambda reduction (here performed by partial execution).

With these arguments added, TALK looks as follows.

%%%% talk is the top-level predicate

talk(\logform) \rightarrow
  s(\logform, nogap).

%%%% Declarative sentences

\[ s(S, \text{GapInfo}) \rightarrow \]
\[ \text{np(VP}^\text{S}, \text{nogap)}, \text{vp(finite,VP,GapInfo)}. \]

- S is the logical form of the entire sentence.

- VP is the logical form of the vp.
%%% Noun phrases

np(NP,nogap) -->
  det(N2^NP),
  n(N1),
  optrel(N1^N2).
np(NP,nogap) -->
  pn(NP).
np((X^S)^S,gap(np,X)) -->
  [].

%%% Verb phrases

vp(Form,X^S,GapInfo) -->
  tv(Form,X^VP),
  np(VP^S,GapInfo).
vp(Form,VP,nogap) -->
  iv(Form,VP).
vp(Form,VP2,GapInfo) -->
  aux(Form/Required,VP1^VP2),
  vp(Required,VP1,GapInfo).
vp(finite,X^S,GapInfo) -->
  [is],
  np((X^P)^exists(X,S&P),GapInfo).

%%% Relative clauses

optrel((X^S1)\(X^S2)) -->
  relpron,
  vp(finite,X^S2,nogap).
optrel((X^S1)\(X^S2)) -->
  relpron,
  s(S2,gap(np,X)).
optrel(N^N) -->
  [].

%%% The dictionary

All the rules in the dictionary pass on an extra argument LF. They are otherwise identical to the rules in the dictionary given earlier.

e. g.

det --> [D], [det(D)].

has become

det(LF) --> [D],[det(D,LF)].
%%% The lexicon

det(every,(X^S1)^*(X^S2)^*all(X,S1=>S2)).
det(a,(X^S1)^*(X^S2)^*exists(X,S1&S2)).
det(an,(X^S1)^*(X^S2)^*exists(X,S1&S2)).

n(man,X^man(X)).
n(woman,X^woman(X)).
n(book,X^book(X)).
n(wine,X^wine(X)).

pn(ivan,ivan).
pn(nero,nero).
pn(cleopatra,cleopatra).

aux(does,finite/nonfinite, VP^VP).
aux(did,finite/nonfinite, VP^VP).

relpron(that).
relpron(who).

tv(love,loves,loved,loved,loving,X^Y^loves(X,Y)).
tv(write,writes,wrote,written,writing,X^Y^writes(X,Y)).
tv(phone,phones,phoned,phoning,X^Y^phones(X,Y)).
tv(drink,drinks,drank,drunk,drinking,X^Y^drinks(X,Y)).

iv(run,runs,ran,run,running,X^runs(X)).

2.4.1 An example of logical form construction

As an introduction to how a sentence is parsed and its logical form simultaneously constructed, we present a trace of the parsing of a simple sentence, while trying to describe what happens. The sentence is *Ioan runs*. To render the trace a little more transparent, we have omitted to follow certain irrelevant subgoals.

The two extra arguments previously mentioned (1.3.3), used to keep track of the sentence being parsed, are visible in the trace below, as are the calls to a built-in predicate named 'C ' that Prolog uses to manipulate the list of words.\(^2\) The numbers that open each line are the goal number and the recursion depth respectively.

\(^2\) The predicate 'C ' is discussed further in 5.1.
The reader is urged to trace through the parsing of some more difficult sentences to get the knack of the intricacies of the logical form construction in TALK. It may also be informative to study [12].
3 Extensions to TALK

Working with TALK it soon becomes apparent that several important language constructions are missing and that it would be nice to add these to TALK. Among them are conjunctions (for np's and vp's alike), genitives, negations and indefinite pronouns. In this chapter we suggest an implementation of each of the above. Naturally, our way of performing these extensions is not the only one and ours may not be the most elegant or efficient solution.

Some of the additions bring about global changes and it might be advisable to refrain from incorporating them, if they aren't really necessary in a particular application. In appendix 1, a complete version with all these additions and some more is to be found.

3.1 Noun phrase conjunctions

3.1.1 A discussion of the problem

By noun phrase conjunctions we mean noun phrases like Ivan and Nero and a woman or a man. It is often desirable to include such constructions in a grammar, since they are very common in English and other human languages.

To illustrate, a sentence with a noun phrase conjunction can look like: Ivan loves Cleopatra and Nero. When this sentence is input to TALK we would like it to have the logical form:

loves(ivan, nero) & loves(ivan, cleopatra).

This implies that the logical form of the transitive verb, $X^\wedge Y^\wedge$loves($X, Y$), has to be mapped over both parts of the conjunction giving:

$X^\wedge$($loves(X, nero) \& loves(X, cleopatra)$).

In the end, Ivan must be reduced into the two verb phrase Logical Forms (LFs) connected by a user defined and-operator (&), instead of the usual single one, finally rendering the form shown above.

3.1.2 The implementation

The two connectives are defined as operators of equal precedence:

:: op(500, xfy, &).
:: op(500, xfy, ∨).
Next, the dictionary and the lexicon are augmented with the following:

%%% The dictionary entry

\textit{conjunct(Conj)} \rightarrow [C], [conjunct(C,Conj)].

%%% The lexicon entries

\textit{conjunct(or,S1^S2*(S1\&S2))}.
\textit{conjunct(and,S1^S2*(S1\&S2)).}

Naturally, the np rules have to be transformed to handle the conjunctive constructions. An ancillary level of np-clauses, np1, that is almost identical to the former np, is added, while np henceforth is either just a np1 or two np1's combined by a conjunction:

%%% The new noun phrase rules

\textit{np(NP,nogap)} \rightarrow
\textit{np1(NP)}.
\textit{np(P^nS,nogap)} \rightarrow
\textit{np1(P^nS1)}.,
\textit{conjunct(Conj)},
\textit{np1(P^nS2)},
\text{reduce(Conj,S1,S2,S)}).
\textit{np(X^nS^nS,gap(np,X))} \rightarrow
[].
\textit{np1(NP)} \rightarrow
\textit{pn(NP)}.
\textit{np1(NP)} \rightarrow
\textit{det(Det)},
\textit{n(N1)},
\textit{optrel(N1,N2)},
\text{reduce(Det,N2,NP)}.

However, these changes are not sufficient. The pn and det rules have also to be replaced due to the fact that partial execution is no longer applicable in every case. Since partial execution does not reduce to canonical form, therefore giving incorrect results here, we have exchanged it with calls to \textit{apply} and \textit{freeze} in the det and pn rules.

The user defined predicate \textit{apply} performs a complete lambda reduction (i.e. carries out internal reductions aswell), while \textit{reduce} only performs a single outermost lambda reduction. \textit{Apply} is defined by the single clause:

\textit{apply(X^nY,A,Y1) :- substitute(Y,X,A,Y1)}.

where \textit{substitute(OldTerm,Old,New,NewTerm)} replaces all occurrences of Old in OldTerm with New yielding NewTerm.\footnote{For a full definition of substitute/4, see [15], p. 140.} When the call to apply is
carried out, all variables must have been instantiated. This is accomplished by freeze, a predefined SICStus Prolog predicate that delays execution of its second argument (a goal) until the first argument (a term) has become instantiated. ¹

%%% Proper nouns

\[ pn(VP\wedge S) \rightarrow \]
\[ \begin{array}{l}
\{ pn(PN,E), \]
\[ freeze(VP,apply(VP,E,S))\}. \]
\[ \end{array} \]

- Apply is blocked until VP is instantiated.

%%% Determiners

\[ det(LF) \rightarrow \]
\[ \begin{array}{l}
\{ s\_det(D), \]
\[ LF=P^\wedge Q^\wedge \exists(X,S1\&S2), \]
\[ freeze(P,apply(P,X,S1), \]
\[ freeze(Q,apply(Q,X,S2))))\}. \]
\[ \end{array} \]

- Singular determiners (e.g. a, an, the)
- P and Q are functions and complete lambda reduction has to be postponed until they are known.

\[ det(LF) \rightarrow \]
\[ \begin{array}{l}
\{ pl\_det(D), \]
\[ LF=P^\wedge Q^\wedge \forall(X,S1=>S2), \]
\[ freeze(P,apply(P,X,S1), \]
\[ freeze(Q,apply(Q,X,S2))))\}. \]
\[ \end{array} \]

- Plural determiners (e.g. every, each)

3.2 Verb phrase conjunctions

3.2.1 A discussion of the problem

Verb phrase conjunctions constitute another form of conjunctions in NL sentences, e.g. Ivan runs or loves Cleopatra. Such sentences exhibit verb phrases consisting of two separate verb phrases combined by a conjunction. The logical form of the sentence above would ideally be:

\[ runs(ivan) \vee loves(ivan,cleopatra). \]

Clearly, Ivan has to be distributed over two verb phrase LFs combined by an or-operator (\(\vee\)). The relatively straightforward implementation of this mapping is described in the next sub-section.

¹ In other Prologs freeze has a different meaning: it creates a copy of a term where all uninstantiated variable have been replaced by constants.
3.2.2 The implementation

Expanding TALK to treat verb phrase conjunctions properly does not give rise to the same amount of hassle as in the case of noun phrase conjunctions. No rules other than those concerning verb phrases have to be rewritten and even those are not subjected to any dramatic changes.

%%% Verb phrases

\[
\begin{align*}
\text{vp}(\text{Form}, \text{VP}, \text{GapInfo}) & \rightarrow \\
& \text{vp1}(\text{Form}, \text{VP}, \text{GapInfo}). \\
\text{vp}(\text{Form}, \text{X}^\text{VP}, \text{GapInfo}) & \rightarrow \\
& \text{vp1}(\text{Form}, \text{X}^\text{VP1}, \text{GapInfo}), \\
& \text{conjunct(Conj)}, \\
& \text{vp1}(\text{Form}, \text{X}^\text{VP2}, \text{GapInfo}), \\
& \{\text{reduce(Conj, VP1, VP2, VP)}\}.
\end{align*}
\]

Similarly to the case of np conjunctions, a new level of \text{vp} is inserted into the grammar, causing the renaming of all former \text{vp} rules to \text{vp1}. Then, in order to ensure that the two \text{vp} LFs in the conjunction will contain the same agent, the bound variables are matched against each other. A special case of lambda reduction later performs the merging of the two \text{vp} lambda expressions into one that binds the variable, \text{X}, when the \text{vp} rule subsequently returns it. The new \text{reduce/4} predicate responsible for the two-fold lambda reduction looks like:

\[
\text{reduce}((\text{Args}1^\text{Expr}, \text{Args}2^\text{Expr}), \text{Args}1, \text{Args}2, \text{Expr}).
\]

3.3 Genitives

3.3.1 A discussion of the problem

The concept of relations in a grammar is made more interesting if nouns naming relations, (e. g. sister) can be combined with genitives. We would then be able to parse sentences looks like:

\emph{Ivan's sister loves Nero}

and obtain the logical form:

\[
\text{exists}(X, \text{sister}(X, \text{ivan}) \& \text{loves}(X, \text{nero})).
\]

Hence, an existentially quantified logical form must be generated somewhere in the grammar. Later on, two lambda expressions (in the example above $X^\text{sister}(X, \text{ivan})$ and $X^\text{loves}(X, \text{nero})$) have to be substituted into the quantified expression and their variables unified with the quantified one.
3.3.2 The implementation

An argument, *Case*, is added to a multitude of clauses for keeping track of a possible genitive construction. *Case* is either *gen* or *not gen* and indicates whether to look for a relation or another genitive (*gen*) or to end the parsing of the noun phrase (*not gen*).

The lexical entries for proper nouns and nouns have to be duplicated to make room for the genitives. For example, *Ivan* is matched by the lexical entry

\[ \text{pn(not gen,ivan,ivan).} \]

(where the second argument is the token and the third the LF), and *Ivan’s* by

\[ \text{pn(gen,ivan’s,ivan).} \]

Moreover, entries for relations have to be added to the grammar. Of course, for generality, these rules also have an argument *Case*.

%%% The dictionary entry for relations between nouns

\[
\text{relational cn(ReI,Case) }\rightarrow \\
\text{[R],} \\
\text{\{relation(ReI,R,ReI)\}.}
\]

%%% A lexicon entry for a relation

\text{relation(not_gen,sister,X,Y \*sister(X,Y)).} \\
\text{relation(gen,’sister’s’,X,Y \*sister(X,Y)).}

The **np** rules must once again be altered. **Np2** rules are now what the **np1** rules used to be. A novel rule **np2_complement** takes care of the genitive and relational constructions. Note the recursion in **np2_complement** that enables multiple genitives (as in *Ivan’s sister’s husband’s* ...).

%%% The changed np rules

\[
\text{np1(NP) }\rightarrow \\
\text{np2(NP1,Case),} \\
\text{np2_complement(NP1,NP,Case).} \quad \text{- If no genitive, NP becomes equal to NP1.}
\]

\[
\text{np2(NP,Case) }\rightarrow \\
\text{pn(NP,Case).} \\
\text{np2(NP,Case) }\rightarrow \\
\text{det(Det),} \\
\text{n(N1,Case),} \\
\text{optrol(N1,N2),} \\
\text{\{reduce(Det,N2,NP)\}.}
\]
np2_complement(NP,NP,notgen) --> [].
np2_complement(NP1,NP,gen) -->
   relational_cn(Rel_CN,Case),
   {combine_gen_relational(NP1,Rel_CN,NP2),
    implicit_det(Det),
    reduce(Det,NP2,NP3)},
   np2_complement(NP3,NP,Case).

% The special reduce
combine_gen_relational((Y^Rel)^S,X^Y^Rel,X^S).

3.4 Negations

3.4.1 A discussion of the problem

Another desirable feature is the recognition and correct parsing of negated sentences, e.g. Ivan doesn't love Cleopatra. The message one wishes to convey with such a sentence is of course the falsity of the corresponding positive sentence.

3.4.2 The implementation

An operator representing negation is declared:

```
:- op(100,fx,~).
```

With this operator, the logical form of the sentence above can be represented:

```
~loves(ivan,cleopatra)
```

The actual negation is performed in the lexical entries for aux with the aid of a predicate negate_vp, that simply adds the ~-operator in front of the vp:

```
negate_vp(X^VP,X^~(VP)).
```

A (negative) aux rule then looks like:

```
aux('doesn't',finite/nonfinite,VP^Not_VP) :-
   freeze(VP,negate_vp(VP,Not_VP)).  \ Don't negate until VP is known!
```

3.5 Dealing with "somebody" and "everybody"

The heading of this section is slightly misleading, since not all of what we introduce are pronouns, let alone indefinite. Nevertheless, their treatment being so similar, we feel it justifies this disrespect for linguistics.
3.5.1 A discussion of the problem

Being able to refer to some (or all) individuals in the universe of discourse is sometimes convenient. The grammar hitherto developed does not allow for expressions like somebody or everyone, but only the qualified some man or every woman &c.

Here, a choice of logical forms has to be made before we start implementing. Consider the sentence Everybody loves Cleopatra. The most apparent alternative for the logical form is perhaps:

\[ \text{all}(X, \text{loves}(X, \text{cleopatra})) \]

but another equally plausible representation is:

\[ \text{all}(X, \text{individual}(X) \Rightarrow \text{loves}(X, \text{cleopatra})) \]

and a case can be made for either.

The first representation seems more natural but has the disadvantage of being somewhat cumbersome to implement, as its logical structure differs from the ones seen so far in TALK. Therefore we have chosen to implement the second, whose logical form follows the norm, but forces us to introduce the pseudo predicate \text{individual}(X).

3.5.2 The implementation

The inclusion of indefinite pronouns into the grammar of TALK does not alter the grammar. In fact, if we view an indefinite pronoun like somebody as a concatenation of a determiner (some) and a noun (body) then we don’t have to extend the grammar rules at all, as they already take care of that kind of construction. We only have to change the pre-processing and expand the lexicon a little.

Before a sentence can be parsed by TALK, it has to be input and converted into a list of words. A predicate \text{read_sentence} deals with this conversion and we extended it to handle the separation of the indefinite pronouns into two words, as can be seen in appendix 1.

Finally, we augment the lexicon with two new "nouns": one and body, both rendering the same logical form, \(X^{\text{individual}}(X)\).

3.6 Conclusions and suggestions

In this chapter we have explored a few ways of extending TALK. These extensions serve a double purpose. Firstly, we hope that they render TALK a more serviceable NL system, one that may be put to actual use in simple AI applications or as a front-end to other kinds of systems. Secondly, we hope to have conveyed a feel for how to tackle the import of grammatical constructions in a DCG.
Inherent in NL research is the fact that no matter what is done, the task of building a perfect NL system will probably never be fulfilled. Of course, it is impossible to foresee the future but most linguists and AI researchers seem to agree on this point. However, TALK, as it stands now, leaves much more to be desired than do some other NL-systems. We suggest here a set of supplementary features that could, in some cases easily and in others with more difficulty, be added to TALK.

* Grammatical number
* Attributes
* Numerals
* Passive sentences
* An ameliorated treatment of copula
* Comparatives
...
4 An Example Application Using TALK

Simple though it may seem, TALK proves quite useful as an interface to various applications. To illustrate this fact, in the present chapter we discuss and implement a complete application that solves a certain kind of logic puzzles: jobs puzzles. When combining TALK with other tools, in our case a theorem prover, it can be seen that only minor application-specific changes are needed. In fact, if the grammar suffices, the only mandatory one is the customization of the lexicon (although other tricks may come in handy).

4.1 A specification of the problem

Our application aims to find the solution to particular problems, an example of which can be seen below:

The leaders problem

Nero, Ivan, Adolf, Nixon and Cleopatra live in a row of houses. They all drink different drinks (gin, campari, beer, wine or coke ), read different authors (Sartre, Singer, Cartland, Ludlum or Sheldon), listen to different music (Wagner, Abba, Stones, Bach or Prince) and have different hobbies (ballet dancing, sailing, drugs, cuisine or tennis playing).

Derive from the following where they live, what they drink, what they read etc.

Ivan's right-hand neighbor drinks wine.
The ballet dancer listens to Bach.
Adolf doesn't drink wine.
Cleopatra or Nero reads Cartland.
... (and so on)

A famous problem of the same kind, the zebra problem, is found in [15].

A system solving such problems should take natural language sentences as input, formalize them and deduce a solution by theorem-proving operations. Thus, TALK has to be modified and subsequently connected to an appropriate theorem prover.
Figure 3. A schematic picture of the system solving logical puzzles

Among the modifications we find an increase of the grammar to accommodate problem-oriented constructions and an enhancement of the lexicon biased by the application domain. Also, the production of logical forms is adapted to diminish the amount of processing needed between TALK and the theorem prover.

4.2 SATCHMO - the theorem prover

In [11], Manthey & Bry present the Prolog implementation of a FOL theorem prover called SATCHMO (SATisfiability CHecking by MOdel generation). We have worked with a version of SATCHMO updated by Mats Carlsson et al. at SICS.

The input to the SICS version, from now on referred to as SATCHMO, is in Clausal Form (CF). In this notation

\[ [a, b, c] \leftarrow [d, e, f]. \]

represents \( d \land e \land f \rightarrow a \vee b \vee c \). For example, an empty consequent corresponds to \textit{false} while an empty antecedent means \textit{true}. Our CF representation of the sentence \textit{The wine drinker plays tennis} looks like:

\[ \text{plays_tennis}(X) \leftarrow \text{drinks_wine}(X). \]

where the range of \( X \) is the set of individuals in the domain. In our world, only one person plays tennis and we can therefore deduce from the above statement that \textit{The tennis player drinks wine}:

\[ \text{drinks_wine}(X) \leftarrow \text{plays_tennis}(X). \]

Implicitly in the formulation of the puzzle lie the facts that each person likes for instance some drink and that it is different from the ones drunk by the others. If we formalize this knowledge into CF, we get what we call the
background axioms for the puzzle. For example, here is a formalization of what we know about the participants' drinking habits:

Nobody drinks more than one beverage:

[]<--[drinks_coke(X),drinks_wine(X)].
[]<--[drinks_campari(X),drinks_wine(X)].
[]<--[drinks_campari(X),drinks_coke(X)].
[]<--[drinks_beer(X),drinks_wine(X)].
[]<--[drinks_coke(X),drinks_beer(X)].
[]<--[drinks_campari(X),drinks_beer(X)].
[]<--[drinks_gin(X),drinks_wine(X)].
[]<--[drinks_gin(X),drinks_coke(X)].
[]<--[drinks_gin(X),drinks_campari(X)].
[]<--[drinks_gin(X),drinks_beer(X)].

Everyone drinks gin or ... or wine:

drinks_gin(X),drinks_campari(X),drinks_coke(X),drinks_beer(X),drinks_wine(X)<--[dom(X)].

Every beverage is drunk by someone:

[drinks_gin(ivan),
  drinks_gin(cleopatra),
  drinks_gin(nero),
  drinks_gin(nixon),
  drinks_gin(adolf)]<--[].

... (likewise for the other beverages)

Having made similar observations for the other properties of ivan & co. as well, we formalize and put them in a file to be merged with the logical forms of the input sentences before the proof procedure commences. This file also contains information about the domain and on what it takes to be someones neighbour. Even this information, i.e. the background axioms, could of course be furnished through the NL interface, but that would take us too far afield and therefore we content ourselves with the manual translation discussed above.
4.3 TALK

In order that TALK will be able to "understand" the sentences in the example, and other equally informative statements, we proceed in the manner of chapter three and add to the grammar: compound nouns (e.g. wine drinker), attributes concerning houses (e.g. the first house), "inhabitive" constructions (e.g. lives in, lives on the right-hand side of) and relations concerning neighborhood (e.g. Ivan's left neighbor). Consult appendix 2 for the implementations.

4.3.1 The new logical forms

In addition, the logical forms produced by the new grammar are more application-oriented and less general than before (although, in the problem domain, logically equivalent), due to the benefits we get from these paraphrases in the form of much simpler intermediate processing.

The new logical forms have another type of quantification than did the old ones: existential quantification with conjunction has been replaced by universal quantification and implication. In our restricted problem domain, it is justified to interpret the singular determiners a and the generically. The former LF for The baker drinks wine:

exists(X, baker(X) & drinks(X, wine)).

transforms to

all(X, baker(X) --> drinks_wine(X)).

where X has the same range as before.

As can be seen above, the logical form of the verb phrase drinks wine has a different appearance, in fact the same as in the examples given in the section about SATCHMO. Evidently, this transformation eases the task of combining the two applications. Not so obviously, perhaps, the logical form of the verb phrase now comes from the constituent noun phrase:

vp1(Form, VP, GapInfo) -->
tv(Form),
np3(VP).

As a consequence, certain noun phrases have to give different logical forms depending on their position in the sentence. If a particular noun phrase succeeds a verb, it has to return another logical form than if it had preceded the same verb, because the verbs no longer produce any logical forms into which the succeeding noun phrases can be substituted. Therefore, the np3 and np4 rules come into existence:

27
\%\%\% Another set of NP rules

np3(NP) \rightarrow np4(NP).
np3(P^S) \rightarrow np4(P^S1),
    conjunct(Conj),
np4(P^S2),
    \{reduce(Conj,S1,S2,S}\}.

np4(D) \rightarrow [house],
    dig(D).
np4(At\_r) \rightarrow [det],
    atr(At\_r),
    [house].
np4(NP) \rightarrow [det],
    n(NP,not\_gen).
np4(NP) \rightarrow pn1(NP).

- e.g. house 1 returns X^lives\_in\_first\_house(X)
- this det only matches a token
- atr gives the LF of the np rule
  e.g X^lives\_in\_green\_house(X)
- the det doesn't return an LF
- a relative clause is not permitted
- same as pn but doesn't freeze or apply

We repeat that the changes to the logical forms of TALK are made as a mere convenience for subsequent processing.

\subsection{4.4 Connecting TALK and SATCHMO}

While TALK produces one form of logical output, SATCHMO requires another form as input, viz. CF. Therefore, a filter has to be designed and fitted between the two. The changes performed in the preceding section somewhat lessen the rather tedious task of constructing this filter. However, the logical forms of TALK must be classified before they can play a part in the deductive process. O'Keefe once designed a program that converts formulas in FOL to clausal form. With only a minimum of changes, his program does exactly what we want. The main predicate \textit{demo/1}, that connects everything into a final application, is shown below.

demo(DomainFile) :-
    talk\_to\_user("junk/file1"),
    identify("junk/file1","junk/file2"),
    clausal\_form\_of\_file("junk/file2","junk/file3"),
    stripped("junk/file3","junk/file4"),
    consult("junk/file4"),
    consult(DomainFile),
    prepare\_db,
    satisfiable(M),!,
    write(M).

- DomainFile is the background axioms
- the NL processing, i.e. TALK
- processing that speeds up SATCHMO
- O'Keefe's utility
- remove erroneous clauses
- consult the result
- consult the background axioms
- prepare a database in Prolog form
- look for a solution
- print it if there was one

The complete program, apart from the clausal form translator and the theorem prover, can be found in appendix 2.
4.5 Running the program

Finally we are able to run an example, the leaders problem given in 4.1. Prompted by the main loop, we supply the following sentences, sufficient to give a unique solution to the puzzle:

ivan's right-hand neighbor drinks wine
the ballet dancer listens to bach
adolf doesn't drink wine
cleopatra or nero reads cartland
the tennis player's left neighbor reads ludlum
nero is an abba fan
the wine drinker's right neighbor cooks
the wagner listener reads sartre
the campari drinker lives in the third house
ivan drinks coke
nero's neighbor doesn't take drugs
nero's neighbor doesn't sail
the singer reader lives in house 5
ivan takes drugs
the ballet dancer reads cartland
the campari drinker cooks
nixon drinks beer
adolf lives on the right-hand side of nixon
ivan listens to stones
the gin drinker lives in the fifth house

TALK translates the sentences to the following logical forms:

all(X,r_neighbor(X,ivan)) => drinks_wine(X).
all(X,dances_ballet(X)) => listens_to_bach(X).
~drinks_wine(adolf).
reads_cartland(cleopatra) or reads_cartland(nero).
all(X,all(Y,plays_tennis(Y => l_neighbor(X,Y))) => reads_ludlum(X)).
listens_to_abba(nero).
all(X,all(Y,drinks_wine(Y) => r_neighbor(X,Y)) => cooks(X)).
all(X,listens_to_wagner(X) => reads_sartre(X)).
all(X,drinks_campari(X) => lives_in_third_house(X)).
drinks_coke(ivan).
all(X,neighbor(X,nero) => ~takes_drugs(X)).

1 Note that this acquires all of Nero's neighbors of drug abuse.
all(X, neighbor(X, nero) => ~sails(X)).
all(X, reads_singer(X) => lives_in_fifth_house(X)).
takes_drugs(ivan).
all(X, dances_ballet(X) => reads_cartland(X)).
all(X, drinks_campari(X) => cooks(X)).
drinks_beer(nixon).
r_neighbor(adolf, nixon).
listens_to_stones(ivan).
all(X, drinks_gin(X) => lives_in_fifth_house(X)).

After a little while the program finds the solution:

lives_in_first_house(ivan)  lives_in_second_house(cleopatra)  lives_in_third_house(nero)
drinks_coke(ivan)            drinks_wine(cleopatra)            drinks_campari(nero)
listens_to_stones(ivan)      listens_to_bach(cleopatra)       listens_to_abba(nero)
takes_drugs(ivan)            dances_ballet(cleopatra)      cooks(nero)
reads_sheldon(ivan)          reads_cartland(cleopatra)       reads_ludlum(nero)
lives_in_fourth_house(nixon) lives_in_fifth_house(adolf)   lives_in_third_house(nero)
drinks_beer(nixon)           drinks_gin(adolf)              drinks_campari(nero)
listens_to_wagner(nixon)     listens_to_prince(adolf)        listens_to_abba(nero)
plays_tennis(nixon)          sails(adolf)                    cooks(nero)
reads_sartre(nixon)          reads_singer(adolf)              reads_ludlum(nero)
5 A Pure TALK

For purposes of another research topic, currently very in-vogue: parallel
evaluation of Prolog, we have developed a pure version of the NL processor
TALK. By "pure" we mean one where the DCG rules have been expanded to
proper Prolog clauses, where all calls to extralogical predicates have been
done away with, and where bound variables are not represented by Prolog
variables.\(^1\) The purity was called for by the nature of the parallel system we
wanted to test, ANDORRA Prolog, currently being developed at the LPS
Laboratory (Logic Programming Systems) at SICS, cf. [8],[9].

We include the following discussion of the purification, because
many points of interest, not only to parallelism, arise in the process.

5.1 The expansion of DCG rules to Prolog clauses

It was stated before (1.3.3) that DCG rules are nothing but "syntactic sugar"
for Prolog clauses, meaning that all DCG rules can equally well be expressed
directly in ordinary Prolog. The present section deals with the corre-
spondence and translation between the two notations.

As we hinted at earlier, each grammar rule hides two extra
arguments; the first representing the input string and the second the
remainder of the input string after the application of the rule. Were these
arguments made explicit, most of the translation task would be completed.

A DCG rule

\[
np(NP) \rightarrow pn(NP).
\]

straightforwardly changes into

\[
np(NP,L0,L) :- pn(NP,L0,L).
\]

In DCG rules with several non-terminals in the consequent, as in

\[
s(S) \rightarrow 
\begin{align*}
& np(VP^S), \\
& vp(VP).
\end{align*}
\]

the list is divided between the rules:

---
\(^1\) Other authors formulate what purity of Prolog programs means slightly differently. In [15] for
instance, Sterling and Shapiro write: Pure Prolog is an approximate realization of the logic programming
computation model on a sequential machine and A pure Prolog program is a logic program, in which
an order is defined for both clauses in the program and goals in the body of the clause.
\[ s(S, L, L) :\]
\[ \text{np}(VP^S, L, L), \]
\[ \text{vp}(VP, L, L). \]

In the case of terminal symbols, calls to the built-in predicate \( C'/3 \) extract elements from the input string. \( C'/3 \), commonly known as \textit{connects}, is defined by the fact

\[ C'(X, L, X, L). \]
and a call

\[ C'(L, X, L, L2) \]
has the declarative reading \( L1 \) is connected to \( L2 \) by the terminal \( X \). For example

\[ \text{copula}(C, \text{copula}) \rightarrow [C, \{\text{copula}(C, \text{copula})\}] \]

becomes

\[ \text{copula}(C, \text{copula}, L, L) :\]
\[ C'(L, C, L), \]
\[ \text{copula}(C, \text{copula}). \]

From the example it should be clear that Prolog calls, priorly within braces, translate to themselves while the bracketed \( C \) turns into a call to \( C' \).

### 5.2 Avoiding calls to extralogical predicates in TALK

Prolog programmers often resort to extralogical predicates (the \textit{cut} and the \textit{call}, for instance)\(^1\), for convenience and occasional necessity. It has long been understood, however, that pure logic programs, i.e. programs excluding extralogical features, are desirable from a mathematical and theoretical point of view.

The TALK system does not rely heavily on the more dubious parts of Prolog, although some extralogical calls solving particular problems are discernible, notably in the \textit{substitute} predicate. The removal of such constructions from TALK, mandatory to get the program running in ANDORRA, results in a new version of \textit{substitute}/4 without cuts and calls to the extralogical \textit{var} predicate:

\[ 1 \text{ Some of the extralogical facilities of Prolog are referred to as metalogical, to distinguish them from the truly extralogical ones, that achieve side effects (e.g. the I/O predicates and predicates for interfacing with the operating system). The metalogical facilities are so called because their semantic domain is that of logical expressions and proofs.} \]
The pure substitute/4

```prolog
substitute(Old,Old,New,New).
substitute(Old,O,N,New) :-
    functor(Old,F,Arity),
    functor(New,F,Arity),
    substitute1(Old,O,N,New,Arity).

substitute1(Old,O,N,New,0).
substitute1(Old,O,N,New,I) :-
    arg(I,Old,OldArg),
    arg(I,New,NewArg),
    substitute(OldArg,O,N,NewArg),
    I1 is I - 1,
    substitute1(Old,O,N,New,I1).
```

N. B. that the `substitute/4` predicate needs all its input arguments to be ground, i.e. without variables, in order to function properly; a restriction easily complied with, as we shall see in the next section.

Due to the way ANDORRA works (determinate and-parallel execution [8]), the freezes become superfluous and can be removed without further ado.

### 5.3 Replacing Prolog variables in the logical forms

For some time it has been customary to use Prolog variables for quantified variables in logical forms produced by DCGs; a comfortable and by no means incorrect way of representing logical variables, in most cases. However, in the clean version developed here, tests on variables are disallowed, by virtue of their extralogical nature. To avoid completely Prolog variables, where not intended, we carry out two steps of refinement.

First we introduce a counter, `gensym(L,R)`, as a parameter to every rule; one that keeps track of the number of variables introduced. `Gensym`'s first argument, `L`, represents the value of the counter at the left end, i.e. the beginning of the string spanned by the rule, while the second represents the value at the right end. Every rule that needs logical variables in its output production increments the counter and passes it on to other rules.

Then a special term, `lvar(L)`, (short for lambda variable), in concert with the `gensym` counter, makes up the new variable mechanism. The parameter of `lvar` takes its value from the `gensym` counter. In this manner, `lvars` replace the Prolog variables in the logical forms. Consequently, the logical variables now consist of a form of constants and we no longer need unification to ensure equality between variables.

The following illustrates the interaction between the rules and the `gensym` counter.
%%% A dictionary entry

\[ n(LF, not\_gen,S0,S, gensym(L,R)) :- \]
\[ 'C'(S0,N,S), \]
\[ n(N,LF, gensym(L,R)). \]

%%% A corresponding lexical entry

\[ n(baker, lvar(L)^{variable}baker(lvar(L)), gensym(L,L+1)). \]

- R in the dictionary rule becomes L + 1, since one new lambda variable is introduced

A more complicated example, perhaps?

%%% The dictionary entry for the det rule

\[ det(LF,S0,S, gensym(L,R)) :- \]
\[ 'C'(S0,D,S), \]
\[ det(D, gensym(L,L1)), \]
\[ R \text{ is } L1 + 1, \]
\[ LF=P\times Q\times \text{exists}(lvar(L),LF1\&LF2), \]
\[ \text{apply}(P,lvar(L),LF1), \]
\[ \text{apply}(Q,lvar(L),LF2). \]

- give a value to L1
- increase R since one new variable is needed
- make sure that P and Q contain this variable

%%% A sample lexical entry

\[ det(a, gensym(L,L)). \]

- no variable is introduced here (R=L)
6 Conclusion

The computer is one of modern man's most essential tools and there are those who hold the view that computers could and should be used to aid man in almost everything he does. However one feels about this, it is important to realize that proper use of computers relies on a smooth man-machine communication. Ideally, this communication is carried out in the user's own language thus removing some of the mystery surrounding computer use. Even though natural language researchers and others are active all over the world attempting to solve the problem of building better and better natural language interfaces, they still have a long way to go towards perfection and there is no telling whether they will ever reach it.

In this thesis we have shown the step-by-step development of a simple natural language interface. Adding new features to our original system we have eventually produced a quite useful natural language interface in its own limited way, which is shown in a working application. Our work has taught us that once you get the knack, many extensions aren't very difficult to make on their own, but putting them all in the same program proves to be quite another affair, as they bring about changes in different directions. Amalgamating it all could easily have resulted in a veritable patchwork, due to the underlying structure of the interface.

In conclusion, our extended system is suitable for some small AI applications but unsuitable for others. Should a more complete system be desired, care must be taken in advance to let the language domain guide the choice of methodology. This choice may very well be different from ours.
Appendix A The extended NL system

% File: ext_talk.pl
% Authors: Anna Bang and Per Lindberger, June 1989
% Based on ideas from
% Pereira and Shieber: Prolog and Natural Language Analysis
% Purpose: Process natural language declarative sentences
% (c) 1989 SICS, Swedish Institute of Computer Science, Stockholm.

:- op(500,xfy,&).
:- op(500,xfy,\).
:- op(520,xfy,=>).
:- op(100,fx,~).

%%% main_loop
%%% =========
%%% The main loop for terminal interaction.
%%% It continually prompts for sentences to be parsed, parses
%%% them and outputs their equivalent logical form.

main_loop :-
    write('>> '),
    read_sentence(Words),
    (quit(Words),nl; 
     (talk(LF,Words,[],)
      numeric(LF,0,M),
      write(LF),nl;
      format("~nl can't do that.\n")(LF)),
    main_loop).

%%% quit
%%% ===
%%% Breaks the main loop
%%% if one of the following tokens
%%% were input: quit, end or bye.

quit([quit]).
quit([end]).
quit([bye]).

%%% talk
%%% ====
%%% Returns the logical form of the sentence, i.e LogForm.

talk(LogForm) -->
    s(LogForm,nogap).

%%% Declarative sentences
%%% =======================

s(S,GapInfo) -->
    np(NP,nogap),
    vp(finite,VP,GapInfo),
    {reduce(NP,VP,S)}.  

36
%% Noun phrases
%% ===========

np(NP,nogap) -->
   np1(NP).
np(P^S,nogap) -->
   np1(P^S1),
   conjunct(Conj),
   np1(P^S2),
   {reduce(Conj,S1,S2,S)}.
np((X^S)^S,gap(np,X)) -->
   [].

np1(NP) -->
   np2(NP1,Case),
   np2_complement(NP1,NP,Case).

np2(NP,Case) -->
   pn(NP,Case).
np2(NP,Case) -->
   det(Det),
   n(N1,Case),
   optrel(N1^N2),
   {reduce(Det,N2,NP)}.

np2_complement(NP,NP,notgen) --> [].
np2_complement(NP,NP,gen) -->
   relational_cn(Rel_CN,Case),
   {combine_gen_relational(NP1,Rel_CN,NP2),
    cheat_det(Det),
    reduce(Det,NP2,NP3)},
   np2_complement(NP3,NP,Case).

%% Verb phrases
%% ===========

vp(Form,VP,GapInfo) -->
   vp1(Form,VP,GapInfo).
vp(Form,X^VP,GapInfo) -->
   vp1(Form,X^VP1,GapInfo),
   conjunct(Conj),
   vp1(Form,X^VP2,GapInfo),
   {reduce(Conj,VP1,VP2,VP)}.
vp(Form1,VP2,GapInfo) -->
   aux(Form1/Form2,VP1^VP2),
   vp(Form2,VP1,GapInfo).
vp1(Form,VP,GapInfo) -->
   tv(Form,TV),
   np(NP,GapInfo),
   {tv_n_p_reduce(TV,NP,VP)}.
vp1(Form,VP,nogap) -->
   iv(Form,VP).
vp1(finite,VP,GapInfo) -->
   copula(Copula),
   np(NP,GapInfo),
   {tv_n_p_reduce(Copula,NP,VP)}.
vp1(finite,VP,GapInfo) -->
ncopula(Copula),
np(NP,GapInfo),
{tv_np_reduce(Copula,NP,VP1),
negate_np(VP1,VP1)).

%%% Relative clauses

***

coprel([X'S1]^([X'S1&X'S2])) -->
  relpron,
  vpv(finite,X'S2,nogap).
coprel([X'S1]^([X'S1&X'S2])) -->
  relpron,
  s(S2,gap(np,X)).
coprel(N^N) -->
  [l].

/*==================================================================*/

Dictionary
/*==================================================================*/

%%% Determiners

%%% ==
det(LF) -->
  [D],
  {s_det(D),
   % Singular determiners yields existential quantification
   LF=P^Q^exists(X,S1&S2),
   freeze(P,(apply(P,X,S1),
           freeze(Q,apply(Q,X,S2))))}.
det(LF) -->
  [D],
  {pl_det(D),
   LF=P^Q^all(X,S1=S2),
   % Plural determiners yields universal quantification
   freeze(P,(apply(P,X,S1),
           freeze(Q,apply(Q,X,S2))))}.

%%% Nouns

%%% ==
n(LF,Case) -->
  [N],
  % Simple nouns like baker, farmer etc.
  {n(Case,N,LF)}.
n(LF,Case) -->
  [P],
  [N],
  % Compound nouns like tennis player, wine drinker etc.
\{pn(notgen,P,Pn),
  n(Case,N,Nn),
  reduce(Nn,Pn,LF)}.

%%% Proper nouns
%%% ============

pn(VP^S,Case) -->
  [PN],
  \{pn(Case,PN,E),
  freeze(VP,apply(VP,E,S))\}.

%%% Auxiliary verbs
%%% ============

aux(Form,LF) -->
  [Aux],
  \{aux(Aux,Form,LF)\}.

%%% Relative pronouns
%%% ============

relpron -->
  [RP],
  \{relpron(RP)\}.

%%% Copula
%%% ======

copula(Copula) -->
  [C],
  \{copula(C,Copula)\}.

copula(Copula) -->
  [C],
  \{copula(C,Copula)\}.

%%% Conjunctions
%%% ============

conjunct(Conj) -->
  [C],
  \{conjunct(C,Conj)\}.

%%% Intransitive verbs
%%% ============

iv(nonfinite,LF) -->
  [IV],
  \{iv(nonfinite,LF)\}.

iv(finite,LF) -->
  [IV],
  \{iv(finite,LF)\}.

iv(finite,LF) -->
  [IV],
  \{iv(finite,LF)\}.

iv(past_participle,LF) -->
IV,  
\{iv(_,_,IV,_,LF)\}.

iv(pres_participle,LF) --> 
\{iv(_,_,IV,_,LF)\}.

%%% Transitive verbs
%%% =============

tv(nonfinite,LF) --> 
\{tv(_,_,TV,_,_,LF)\}.

tv(finite,LF) --> 
\{tv(_,_,TV,_,_,LF)\}.

tv(finite,LF) --> 
\{tv(_,TV,_,_,_,LF)\}.

tv(nonfinite,LF) --> 
\{tv(_,_,TV,_,_,LF)\}.

tv(past_participle,LF) --> 
\{tv(_,_,TV,_,_,LF)\}.

tv(pres_participle,LF) --> 
\{tv(_,_,TV,_,_,LF)\}.

...
[to],
{xxtv(_,_,_,TV,LF)}.

%%%% Relations between nouns
%%%% -------------------------

relational_cn(Rel,Case) -->
 [R],
 {relation(Case,R,Rel)}.

/*================================================================================================*/

Lexical Items

relational {notgen,neighbor,X^nY^nneighbor(X,Y)).
relational {gen,'neighbor's',X^nY^nneighbor(X,Y)).
relational {notgen,mother,X^nY^mother(X,Y)).
relational {gen,'mother's',X^nY^mother(X,Y)).
relational {notgen,father,X^nY^nfather(X,Y)).
relational {gen,'father's',X^nY^nfather(X,Y)).
relational {notgen,brother,X^nY^brother(X,Y)).
relational {gen,'brother's',X^nY^brother(X,Y)).
relational {notgen,sister,X^nY^sister(X,Y)).
relational {gen,'sister's',X^nY^sister(X,Y)).
relational {notgen,wife,X^nY^nwife(X,Y)).
relational {gen,'wife's',X^nY^nwife(X,Y)).
relational {notgen,husband,X^nY^nhusband(X,Y)).
relational {gen,'husband's',X^nY^nhusband(X,Y)).
relational {notgen,son,X^nY^son(X,Y)).
relational {gen,'son's',X^nY^son(X,Y)).
relational {notgen,daughter,X^nY^daughter(X,Y)).
relational {gen,'daughter's',X^nY^daughter(X,Y)).

s_det(a).
s_det(the).
s_det(an).
s_det(some).
pl_det(every).
pl_det(each).

cheat_det(LF) :-
    LF = P^Q^exists(X,S1&S2),
    freeze(P,(apply(P,X,S1)),
    freeze(Q,apply(Q,X,S2))).

pn(notgen,ivan,ivan).
pn(notgen,nero,nero).
pn(notgen,cleopatra,cleopatra).
pn(notgen,nixon,nixon).
pn(notgen,adolf,adolf).
pn(notgen,coke,coke).
pn(notgen,wine,wine).
pn(notgen,campari,campari).
pn(notgen,beer,beer).
pn(notgen,scotch,scotch).
pn(notgen,gin,gin).
pn(notgen,sartre,sartre).
pn(notgen,wagner,wagner).
pn(notgen,abba,abba).
 pn(notgen,stones,stones).
 pn(notgen,bach,bach).
 pn(notgen,prince,prince).
 pn(notgen,singer,singer).
 pn(notgen,ludlum,ludlum).
 pn(notgen,carland,carland).
 pn(notgen,sheldon,sheldon).
 pn(notgen,tennis,tennis).
 pn(notgen,bridge,bridge).
 pn(notgen,golf,golf).
 pn(notgen,chess,chess).
 pn(gen,'ivan's',ivan).
 pn(gen,'nero's',nero).
 pn(gen,'cleopatra's',cleopatra).
 pn(gen,'nixon's',nixon).
 pn(gen,'adolf's',adolf).

n(notgen,person,X^person(X)).
 n(notgen,baker,X^baker(X)).
 n(notgen,dentist,X^dentist(X)).
 n(notgen,nurse,X^nurse(X)).
 n(notgen,farmer,X^farmer(X)).
 n(notgen,bartender,X^bartender(X)).
 n(notgen,plumber,X^plumber(X)).
 n(notgen,doctor,X^doctor(X)).
 n(notgen,body,X^individual(X)).
 n(notgen,one,X^individual(X)).
 n(notgen,man,X^man(X)).
 n(notgen,woman,X^woman(X)).
 n(notgen,student,X^student(X)).
 n(notgen,drinker,X^drinks(Y,X)).
 n(notgen,listener,X^listens(Y,X)).
 n(notgen,reader,X^reads(Y,X)).
 n(notgen,fan,X^likes(Y,X)).
 n(notgen,player,X^plays(Y,X)).
 n(notgen,book,X^book(X)).
 n(notgen,novel,X^novel(X)).
 n(notgen,paper,X^paper(X)).
 n(notgen,beverage,X^beverage(X)).
 n(notgen,cocktail,X^cocktail(X)).
 n(notgen,record,X^record(X)).
 n(notgen,single,X^single(X)).
 n(notgen,house,X^house(X)).
 n(notgen,bungalow,X^bungalow(X)).
 n(gen,'person's',X^person(X)).
 n(gen,'baker's',X^baker(X)).
 n(gen,'dentist's',X^dentist(X)).
 n(gen,'nurse's',X^nurse(X)).
 n(gen,'farmer's',X^farmer(X)).
 n(gen,'bartender's',X^bartender(X)).
 n(gen,'plumber's',X^plumber(X)).
 n(gen,'doctor's',X^doctor(X)).
 n(gen,'body's',X^individual(X)).
 n(gen,'one's',X^individual(X)).
 n(gen,'man's',X^man(X)).
 n(gen,'woman's',X^woman(X)).
n(gen,'student's',X^Y^student(X)).
n(gen,'drinker's',X^Y^drinks(Y,X)).
n(gen,'listener's',X^Y^listens(Y,X)).
n(gen,'reader's',X^Y^reads(Y,X)).
n(gen,'fan's',X^Y^likes(Y,X)).
n(gen,'player's',X^Y^plays(Y,X)).

aux(do,finite/nonfinite, VP^VP).
aux(does,finite/nonfinite, VP^VP).
aux('doesn't',finite/nonfinite, VP^Not_VP) :-
    freeze(VP,negate_vp(VP,Not_VP)).
aux('don't',finite/nonfinite, VP^Not_VP) :-
    freeze(VP,negate_vp(VP,Not_VP)).
aux('is',finite/pres_participle, VP^VP).
aux('is',finite/pres_participle, VP^Not_VP) :-
    freeze(VP,negate_vp(VP,Not_VP)).

reipron(that).
reipron(who).
reipron(whom).

copula(is,X^Y^(Y=X)).
copula(are,X^Y^(Y=X)).

copula('isn't',X^Y^(Y=X)).

conjunct(or,S1^S2^S1\S2).
conjunct(and,S1^S2^S1\S2).

iv(bake,bakes,baked,baked,baking,X^baker(X)).
iv(farm,farms,farmed,farmed,farming,X^farmer(X)).

tv(drink,drinks,drank,drunk,drinking,X^Y^drinks(X,Y)).
tv(like,likes,liked,liking,X^Y^likes(X,Y)).
tv(love,loves,loved,loving,X^Y^loves(X,Y)).
tv(read,reads,read,reading,X^Y^reads(X,Y)).
tv(play,plays,played,playing,X^Y^plays(X,Y)).
tv(inhabit,inhabits,inhabited,inhabiting,X^Y^inhabits(X,Y)).

xxtv(live,lives,lived,lived,living,X^Y^lives(X,Y)).

read_sentence(W) :-
    get0(C),
    read_sentence(C,W).

read_sentence(C,[]) :-
    newline(C),!.
read_sentence(C,[]) :-
    period(C),!,
    get0(NL),
    newline(NL).
read_sentence(Ch,W) :-
    space(Ch),!,
    get0(C),
    read_sentence(C,W).
read_sentence(Ch,Sent) :-
    read_word(Ch,Chs,Next),
if((pre_suffix(DetChs,Chs,RestChs),
name(Det,DetChs),
det(Det)),
(name(Rest,RestChs),
W=[Det,Rest]),
(name(Word,Chs),
W=[Word])),
read_sentence(Next,Ws),
append(W,Ws,Sent).

read_word(C,[],C) :-
space(C),!.
read_word(C,[],C) :-
newline(C),!.
read_word(C,[],C) :-
period(C),!.
read_word(Ch,[LCh|Chs],Last) :-
(Ch>64 , Ch<91 -> lower_case(Ch,LCh);
Ch=LCh),
goto(Next),
read_word(Next,Chs,Last).

append([],Xs,Xs).
append([X|Xs],Ys,[X|Zs]) :-
append(Xs,Ys,Zs).

pre_suffix([],Ys,Ys).
pre_suffix([X|Xs],[X|Ys],Zs) :-
pre_suffix(Xs,Ys,Zs).

det(some).
det(every).

lower_case(UC,LC):-
LC is UC + 32.

space(32).

newline(10).

period(46).

%%% Real lambda-reduction!!

apply(X^Y,A,Y1) :- substitute(Y,X,A,Y1).

substitute(Var,X,A,A) :- Var == X,!.
substitute(Var,X,A,Var) :- var(Var),!.
substitute(Atom,X,A,Atom) :- atomic(Atom),!.
substitute(Term,X,A,Term1) :-
  functor(Term,F,N),
  functor(Term1,F,N),
  substitute1(Term,X,A,Term1,N).

substitute1(Term,X,A,Term1,0).
substitute1(Term,X,A,Term1,1) :-
  arg(1,Term,Arg),
Appendix B The puzzle solving program

B.1 The top level program

demo(File) :-
   talk_to_user('junk/file1'),
   identify('junk/file1','junk/file2'),
   clausal_form_of_file('junk/file2','junk/file3'),
   stripped('junk/file3','junk/file4'),
   copy_into('junk/file4',File),
   consult('junk/file4'),
   prepare_db,
   satisfiable(M),!,
   write(M).

talk_to_user(File):-
   tell(File),
   talk,
   told.

talk:-
   display('>>> '),
   read_sentence(Words),
   process_sentence(Words).

process_sentence([stop]):- !.
process_sentence([quit]):- !.
process_sentence([end]):- !.
process_sentence(Words):- (talk(LF,Words,[]),
   format('""w."n",[LF]),!;
   error,!),
   nl,
   talk.

error: :-
   display('I can't do that.'),
   typout(10).

identify(File1,File2):-
   see(File1),
   tell(File2),
   read(T),
   id(T),
   told,
   seen.

id(end_of_file).

id(Term):- 
   name(Term,L),
(member(126,L) -> write(Term);
insert_l(L,L1),
nname(Term1,L1),
write(Term1)),
format("--n",[]),
read(NextTerm),
id(NextTerm).

stripped(File1,File2):-
see(File1),
tell(File2),
read(Clause),
skip_clause(Clause),
told,
seen.

skip_clause(end_of_file).
skip_clause(Clause):-
nname(Clause,L),
(sublist("neighbor",L),
sublist("<-[",L),!;
format("~w.-n",[Clause]),!),
read(NextClause),
skip_clause(NextClause).

copy_into(ToFile,FromFile):-
see(FromFile),
open(ToFile,append,FS),
set_output(FS),
copy_term,
close(FS),
seen.

copy_term:-
read(T),
copy(T).

copy(end_of_file):- !.
copy(T):-
format("~w.-n",[T]),
copy_term.

sublist(S,L):-
conc(L1,L2,L),
conc(S,L3,L2).

conc([],L,L).
conc([X|L1],L2,[X|L3]):-
conc(L1,L2,L3).

member(X,[X|L]):- !.
member(X,[L|L]):-
member(X,L).

insert_l([],[]).
insert_l([1,62|Xs],[60,61,62|Ys]) :-
insert_l(Xs,Ys).
insert_l([X|Xs],[Y|Ys]) :-
   insert_l(Xs,Ys).

read_sentence(W) :-
   get0(C),
   read_sentence(C,W).

read_sentence(C,[ ]) :-
   newline(C),!.
read_sentence(C,[ ]) :-
   period(C),!,
   get0(NL),
   newline(NL).
read_sentence(Ch,W) :-
   space(Ch),!,
   get0(C),
   read_sentence(C,W).
read_sentence(Ch,Ws) :-
   read_word(Ch,Chs,Next),
   name(Ch,Chs),
   read_sentence(Next,Ws).

read_word(C,[ ],C) :-
   space(C),!.
read_word(C,[ ],C) :-
   newline(C),!.
read_word(C,[ ],C) :-
   period(C),!.
read_word(Ch,[L|Chs],Last) :-
   (Ch>64, Ch<91 -> lower_case(Ch,LCh);
   Ch=LCh),
   get0(Next),
   read_word(Next,Chs,Last).

lower_case(UC,LC) :-
   **LC** is **UC** + 32.

space(32).

newline(10).

period(46).

B.2 The natural language processing part

% File: demo_talk.pl
% Authors: Anna Bång and Per Lindberger, June 1989
% Purpose: Process natural language declarative sentences specific for the application.
% (c) 1989 SICS, Swedish Institute of Computer Science, Stockholm.

:- op(500,xfy, and ).
:- op(500,xfy, or ).
:- op(520,xfy, => ).
:- op(100,fx, ~ ).
talk(LogForm) -->
   s(LogForm,nogap).

s(S,GapInfo) -->
   np(NP,GapInfo),
   vp(finite,VP,GapInfo),
   {reduce(NP,VP,S)}.

%%% Noun phrases

np(NP,nogap) -->
   np1(NP).
np(P^S,nogap) -->
   np1(P^S1),
   conjunct(Conj),
   np1(P^S2),
   {reduce(Conj,S1,S2,S)}.
np((X^S)^S,gap(np,X)) -->
   [].

np1(NP) -->
   np2(NP,Case),
   np2_complement(NP1,NP,Case).

np2_complement(NP,NP,notgen) --> [].
np2_complement(NP1,NP,gen) -->
   relational_crt(Recl_CN,Case),
   {combine_gen_relnal(NP1,Recl_CN,NP2),
    cheat_dct(Det),
    reduce(Det,NP2,NP3)},
   np2_complement(NP3,NP,Case).

np2(NP,Case) -->
   np(NP,Case).
np2(NP,Case) -->
   det(Det),
   n(N1,Case),
   optrel(N1^N2),
   {reduce(Det,N2,NP)}.
combine_relnal((Y^Rel)^S,X^Y^Rel,X^S).

np3(NP) --> np4(NP).
np3(P^S) -->
   np4(P^S1),
   conjunct(Conj),
   np4(P^S2),
   {reduce(Conj,S1,S2,S)}.

np4(D) -->
   [house],
   dig(D).
np4(Atr) -->
   det,
   atr(Atr),
   [house].
np4(NP) -->
   det,
n(NP, notgen).
np4(NP) ->
  pn1(NP).

%%% Verb phrases

negate_vp(X^VP, X^(~VP)).

vp(Form, VP, GapInfo) -->
  vp1(Form, VP, GapInfo).

vp(Form, X^VP, GapInfo) -->
  vp1(Form, X^VP1, GapInfo),
  conjunct(Conj),
  vp1(Form, X^VP2, GapInfo),
  {reduce(Conj, VP1, VP2, VP)}.

vp(Form1, VP2, GapInfo) -->
  aux(Form1/Form2, VP1^VP2),
  vp(Form2, VP1, GapInfo).

vp1(Form, VP, GapInfo) -->
  tv(Form),
  np3(VP).

vp1(Form, VP, GapInfo) -->
  neighbor_tv(Form, LF),
  np3(NP),
  {inner_reduce(LF, NP, VP)}.

vp1(Form, VP, nogap) -->
  iv(Form, VP).

vp1(Form1, VP2, GapInfo) -->
  aux(Form1/Form2, VP1^VP2),
  vp1(Form2, VP1, GapInfo).

vp1(finite, VP, GapInfo) -->
  copula,
  np3(VP).

vp1(finite, VP, GapInfo) -->
  ncopula,
  np3(VP1),
  {negate_vp(VP1, VP)}.

%%% Relative clauses

optrel((X^S1)^(X^(('S1⇒S2) ))) -->
  relpron,
  vp(finite, X^S2, nogap).

optrel((X^S1)^(X^(('S1⇒S2)))) -->
  relpron,
  s(S2, gap(np, X)).

optrel(N^N) -->
  [].

/*==========================================================================

Dictionary
==========================================================================*/

det --> [D], {det(D)}.

det(LF) -->
  [D],
  {det(D),
LF = P ⊗ all(X, S1 => S2),
freeze(P, apply(P, X, S1),
freeze(Q, apply(Q, X, S2)))

n(LF, Case) -> [N], {n(N, LF, Case)}.
n(Pn, Case) ->
[P],
[N],
{pn1(P, Pn),
n1(N, Case)}.

pn(VP*S, Case) ->
[PN],
{pn(PN, E, Case),
freeze(VP, apply(VP, E, S))}.

pn1(PN) ->
[P],
{pn1(P, PN)}.

pn1(PN) ->
[P],
{pn(P, PN, Case)}.

dig(Dig) ->
[D],
{dig(D, Dig)}.

aux(Form, LF) ->
[Aux],
{aux(Aux, Form, LF)}.

relpron ->
[RP],
{relpron(RP)}.

copula ->
[C],
{copula(C)}.

ncopula ->
[C],
{ncopula(C)}.

conjunct(Conj) ->
[C],
{conjunct(C, Conj)}.

attr(LF) ->
[A],
{attr(A, LF)}.

iv(nonfinite, LF) ->
[IV],
{iv(IV, __, __, LF)}.
iv(finite, LF) ->
[IV],
{iv(__, IV, __, LF)}. % E.g. Nero bakes
iv(pres_participle,LF) -->
  [IV],
  {iv(_,IV,LF)}.

tv(nonfinite) -->
  [TV],
  {trv(TV,_)}.  % E.g. Nero drinks scotch

tv(finite) -->
  [TV],
  {trv(_,TV,_)}.  % E.g. Nero lives in the first house.

tv(pres_participle) -->
  [TV],
  {trv(_,TV)}.%

tv(nonfinite) -->
  [TV],
  [in],
  {xtv(TV,_)}.%

tv(finite) -->
  [TV],
  [in],
  {xtv(_,TV,_)}.%

tv(pres_participle) -->
  [TV],
  [in],
  {xtv(_,TV)}.%

tv(nonfinite) -->
  [TV],
  [to],
  {xtv(TV,_)}.%

tv(finite) -->
  [TV],
  [to],
  {xtv(_,TV,_)}.%

tv(pres_participle) -->
  [TV],
  [to],
  {xtv(_,TV,_)}.%

neighbor_tv(nonfinite,LF) -->
  [live],
  [on],
  [the],
  side1(LF),
  [side],
  [of].

neighbor_tv(finite,LF) -->
  [lives],
  [on],
  [the],
  side1(LF),
  [side],
  [of].

neighbor_tv(pres_participle,LF) -->
  [living],
  [on],
  [the],
  side1(LF),
  [side],
  [of].

% E.g. Nero lives on the right side of Adolf.
neighbor_tv(finite,LF) ->
  [is],
  [the],
  side(LF,notgen),
  [of].

relational_cn(Side,Case) -> side(Side,Case).

side1(Side) -> [S], {side(S,Side)}.

side(X^Y^neighbor(X,Y),Case) -> neighbor(Case).
side(Side,Case) ->
  [S],
  neighbor(Case),
  {side(S,Side)}.

% E.g. Nero is the left-hand neighbor of Adol.

/* ==-----------------------------------------------------------------*/

Lexical Items
/* ==-----------------------------------------------------------------*/

side("right-hand",X^Y^r_neighbor(X,Y)).
side("left-hand",X^Y^l_neighbor(X,Y)).
side(right,X^Y^r_neighbor(X,Y)).
side(left,X^Y^l_neighbor(X,Y)).

neighbor(notgen) -> [neighbor].
neighbor(gen) -> ["neighbor"s].

det(a).
det(the).
det(an).

cheat_det(LF) :- /* This is new, but will be needed anyhow */
  LF = P^Q^all(X,S1=S2),
  freeze(P,apply(P,X,S1),
        freeze(Q,apply(Q,X,S2))).

pn(dolly,dolly,notgen).
pn(kent,kent,notgen).
pn(bruce,bruce,notgen).
pn(ivan,ivan,notgen).
pn(nero,nero,notgen).
pn(cleopatra,cleopatra,notgen).
pn(nixon,nixon,notgen).
pn(adorf,adorf,notgen).
pn("dolly"s,dolly,gen).
pn("kent"s,kent,gen).
pn("bruce"s,bruce,gen).
pn("ivan"s,ivan,gen).
pn("nero"s,nero,gen).
pn("cleopatra"s,cleopatra,gen).
pn("nixon"s,nixon,gen).
pn("adorf"s,adorf,gen).

pn1(wine,X^drinks_wine(X)).
pn1(campari,X^drinks_campari(X)).
pn1(beer,X^drinks_beer(X)).
pn1(scotch,X^drinks_scotch(X)).
pn1(gin,X^drinks_gin(X)).
pn1(cointreau,X\*drinks_cointreau(X)).
pn1(coke,X\*drinks_coke(X)).
pn1(water,X\*drinks_water(X)).
pn1(wagner,X\*listens_to_wagner(X)).
pn1(abba,X\*listens_to_abba(X)).
pn1(stones,X\*listens_to_stones(X)).
pn1(bach,X\*listens_to_bach(X)).
pn1(prince,X\*listens_to_prince(X)).
pn1(sartre,X\*reads_sartre(X)).
pn1(singer,X\*reads_singer(X)).
pn1(ludum,X\*reads_ludum(X)).
pn1(cartland,X\*reads_cartland(X)).
pn1(sheldon,X\*reads_sheldon(X)).
pn1(tennis,X\*plays_tennis(X)).
pn1(bridge,X\*plays_bridge(X)).
pn1(golf,X\*plays_golf(X)).
pn1(chess,X\*plays_chess(X)).
pn1(drugs,X\*takes_drugs(X)).
pn1(drug,X\*takes_drugs(X)).
pn1(ballet,X\*dances_ballet(X)).
pn1(yachting,X\*sails(X)).

n(neighbor,X\*Y\*neighbor(X,Y),notgen).
n(cook,X\*cooks(X),notgen).
n(baker,X\*baker(X),notgen).
n(dentist,X\*dentist(X),notgen).
n(nurse,X\*nurse(X),notgen).
n(farmer,X\*farmer(X),notgen).
n(bartender,X\*bartender(X),notgen).
n(plumber,X\*plumber(X),notgen).
n(doctor,X\*doctor(X),notgen).
n(sailor,X\*sails(X),notgen).
n('cook's',X\*cooks(X),gen).
n('baker's',X\*baker(X),gen).
n('dentist's',X\*dentist(X),gen).
n('nurse's',X\*nurse(X),gen).
n('farmer's',X\*farmer(X),gen).
n('bartender's',X\*bartender(X),gen).
n('plumber's',X\*plumber(X),gen).
n('doctor's',X\*doctor(X),gen).
n('sailor's',X\*sails(X),gen).

n1(drinker,notgen).
n1(listener,notgen).
n1(reader,notgen).
n1(fan,notgen).
n1(player,notgen).
n1(dancer,notgen).
n1(addict,notgen).
n1('drinker's',gen).
n1('listener's',gen).
n1('reader's',gen).
n1('fan's',gen).
n1('player's',gen).
n1('addict's',gen).
n1('dancer's',gen).

dig(1,X\*lives_in_first_house(X)).
dig(2,X\lives_in_second_house(X)).
dig(3,X\lives_in_third_house(X)).
dig(4,X\lives_in_fourth_house(X)).
dig(5,X\lives_in_fifth_house(X)).
dig(one,X\lives_in_first_house(X)).
dig(two,X\lives_in_second_house(X)).
dig(three,X\lives_in_third_house(X)).
dig(four,X\lives_in_fourth_house(X)).
dig(five,X\lives_in_fifth_house(X)).

aux(does,finite/nonfinite, VP^VP).
aux(do,finite/nonfinite, VP^VP).
aux(\text{"doesn’t"}, finite/nonfinite, VP^Not VP) :-
freeze(VP,negate vp(VP,Not VP)).
aux(\text{"don’t"}, finite/nonfinite, VP^Not VP) :-
freeze(VP,negate vp(VP,Not VP)).
aux(is,finite/pres_participle,VP^VP).
aux(\text{"isn’t"}, finite/pres_participle, VP^Not VP) :-
freeze(VP,negate vp(VP,Not VP)).

repron(that).
repron(who).
repron(whom).

copula(is).
copula(are).

copula(\text{"isn’t"}).

conjunct(or,S1^S2\{S1 or S2\}).
conjunct(and,S1^S2\{S1 and S2\}).

attr(green,X\lives_in_green_house(X)).
attr(red,X\lives_in_red_house(X)).
attr(blue,X\lives_in_blue_house(X)).
attr(white,X\lives_in_white_house(X)).
attr(pink,X\lives_in_pink_house(X)).
attr(yellow,X\lives_in_yellow_house(X)).
attr(black,X\lives_in_black_house(X)).
attr(purple,X\lives_in_purple_house(X)).
attr(first,X\lives_in_first_house(X)).
attr(second,X\lives_in_second_house(X)).
attr(third,X\lives_in_third_house(X)).
attr(fourth,X\lives_in_fourth_house(X)).
attr(fifth,X\lives_in_fifth_house(X)).

itv(bake,bakes,baking,X\lives_baker(X)).
itv(farm,farms,farming,X\lives_farmer(X)).
itv(nurse,nurses,nursing,X\lives_nurse(X)).
itv(sail,sails,sailing,X\lives_sails(X)).
itv(cook,cooks,cooking,X\lives_cook(X)).

trv(drink,drinks,drinking).
trv(like,likes,liking).
trv(read,reads,reading).
trv(play,plays,playing).
trv(inhabit,inhabits,inhabiting).
trv(dance,dances,dancing).
trv(take,takes,taking).
xtv(live,lives,living).
xxtv(listen,listens,listening).

%%% Real lambda-reduction!!

apply(X^Y,A,Y1) :- substitute(Y,X,A,Y1).

substitute(Var,X,A,A) :- Var == X,l.
substitute(Var,X,A,Var) :- var(Var),l.
substitute(Atom,X,A,Atom) :- atomic(Atom),l.
substitute(Term,X,A,Term1) :-
    functor(Term,F,N),
    functor(Term1,F,N),
    substitute1(Term,X,A,Term1,N).
substitute1(Term,X,A,Term1,0).
substitute1(Term,X,A,Term1,l) :-
    arg(l,Term,Arg),
    arg(l,Term1,Arg1),
    substitute(Arg,X,A,Arg1),
    l1 is l - 1,
    substitute1(Term,X,A,Term1,l1).

reduce(Args^Expr,Args,Expr).
reduce(Args1^Args2^Expr,Args1,Args2,Expr).
tv_np_reduce(X^VP,VP^S,X^S).
inner_reduce(X^Y^S,Y,X^S).
References


